TURING 图灵原创

Unity API解析

陈泉宏◎编著

- 理解Unity核心API
- 快速提升开发能力



人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

作者简介



陈泉宏 硕士，毕业于北京工业大学。2010年开始从事Android应用程序开发，2011年开始研究Unity，是著名音乐识别软件“音乐雷达”的开发者之一，曾负责中国科学技术馆虚拟装配项目的主要开发工作。

TURING 图灵原创

Unity API 解析

陈泉宏 ◎ 编著

人 民 邮 电 出 版 社
北 京

图书在版编目（CIP）数据

Unity API解析 / 陈泉宏编著. -- 北京 : 人民邮电
出版社, 2014.9
（图灵原创）
ISBN 978-7-115-36651-1

Ⅰ. ①U… Ⅱ. ①陈… Ⅲ. ①游戏程序－程序设计
Ⅳ. ①TP311.5

中国版本图书馆CIP数据核字(2014)第171975号

内 容 提 要

本书挑选了 Unity 引擎里一些核心 API 类，例如 Object、GameObject、Rigidbody、Transform、Camera、Quaternion、Vector3 等进行了详细的功能注解，注解内容包括 API 的使用方法、算法分析、边界条件、参数间的制约关系及注意事项等，特别是对很多功能相近或使用方法相似的 API 进行了较为详细的比较说明。

本书适用于对 Unity 有一定了解的入门开发人员，也可作为 Unity 开发者的参考手册。

◆ 编　　著　陈泉宏
责任编辑　王军花
执行编辑　张　霞
责任印制　焦志炜

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　　　　印刷

◆ 开本：800×1000　1/16
印张：16
字数：409千字　　　　2014年 9 月第 1 版
印数：1－3 000册　　　2014年 9 月北京第 1 次印刷

定价：49.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京崇工商广字第 0021 号

前　言

从我开始接触 Unity 到现在已经两年有余。记得刚开始学习 Unity 的时候甚是惊喜，一是因为 Unity 具有强大的多平台移植功能，随着移动互联时代的到来，该功能越来越顺应时代潮流；二是因为这个引擎容易上手，3 个月的时间足以让一个新手对这套引擎有全面的了解。然而，随着时间的推移，我越来越感觉自己的能力提升似乎遇到了瓶颈。在实际的项目开发过程中，总有一种放不开手脚的感觉，就像自己手里明明有一把屠龙刀，却无法发挥出这把刀真正的威力。Unity 是个入门快、提高难的游戏引擎，想提升能力，至少需要越过 3 道坎：第一道坎就是对 API 的积累，对 API 的合理利用不仅可以减轻自己的编码负担，而且往往可以提高程序的运行效率；第二道坎是 shader 编程，想要做出一款精品游戏往往需要有高效 shader 的支持；最后一道坎就是综合能力了，一款产品的制作除了功能编程外，往往会涉及很多其他领域，例如产品架构、UI 交互设计、模型制作等，作为主要的编程人员，对其他相关领域的了解程度往往会影响到产品的制作直至最后的产品体验。

目前，Unity 引擎的 API 有好几千个，并且随着 Unity 版本的更新，API 的数量还会不断增长。对 API 的熟悉程度直接影响着程序的开发效率，熟悉 API 也成了新手进阶的必经之路。尽管官方给出了较为丰富的 API 文档，然而这并不能满足实际开发的需要，因为官方给出的 API 解释往往只描述了相应 API 的主要功能，缺少对其边界条件的说明和 API 的算法解释。要知道，在实际开发中，必须要明确 API 参数的使用范围，否则往往会出现一些无法预料的状况。于是我决定自己来研究。对 API 的研究几乎占用了我所有的业余时间，在这期间我遇到了各种各样的难题，解决后做了详细的笔记。后来，当笔记积累了厚厚一本的时候，我意识到，也许应该分享出来，使更多的开发者在这上面少花费些时间。于是便有了这本书，希望能对开发者有所帮助。

阅读说明

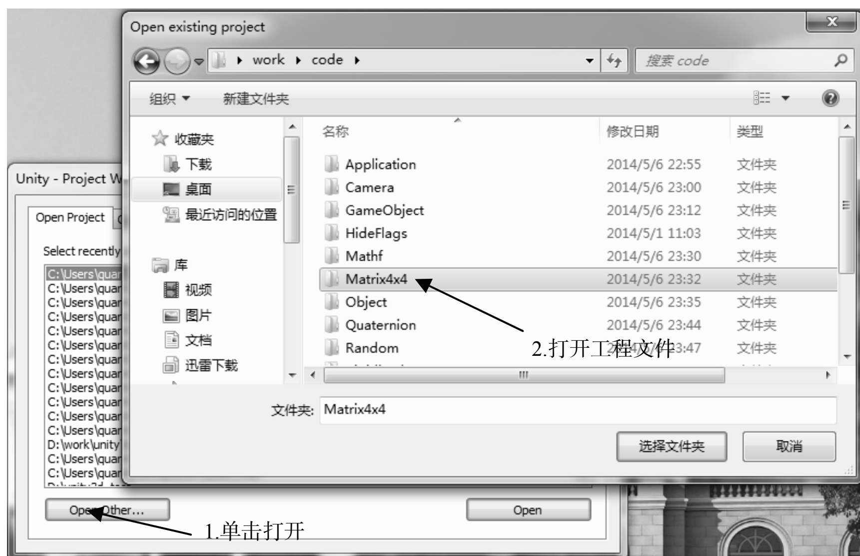
在本书中，以下专业用语的意义是相同的：“变量”与“属性”，“函数”与“方法”，“归一化”与“单位化”。

本书对 API 的描述主要分为 3 个部分：“基本语法”、“功能说明”和“实例演示”。其中，“基本语法”中也包括对 API 参数的简单说明；“功能说明”是对 API 功能的详细介绍，例如使用的注意事项、边界条件、算法分析等；“实例演示”主要是结合实际例子来说明 API 的使用，为了节省篇幅，有些示例代码中会涉及多个 API 的说明。

除了对单个API进行解释之外,本书还对一些功能相近或容易混淆的API进行了相应的解释。另外，书中所涉及的 API 均基于 Windows 操作系统下 Unity 4.3 版本进行了测试。

源码说明

本书所有“实例演示”的源码都可以在图灵社区本书主页(http://www.ituring.com.cn/book/1474)免费注册下载。书中的每个类为一个单独工程，每个工程中场景的命名规则是：API 名字_unity。如果想查看某个 API 的示例代码，只要去 API 所在的类对应的工程文件里打开 API 名字对应的场景即可。例如，打开 Matrix4x4 类中 MultiplyVector 的实例演示的过程如下所示。





致　　谢

这本书得以出版，首先要感谢父母和小妹对我的理解和支持，其次要感谢那些在 API 研究过程中为我解惑的朋友们，最后要特别感谢图灵的编辑王军花和张霞为本书出版提供的真诚指导。

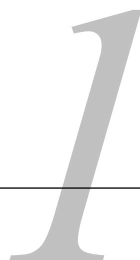
由于时间和能力有限，本书只对 Unity 中的 14 个类进行了讲解和说明，难免有不足或疏漏之处，希望能够抛砖引玉，与大家共同研讨。如有疑问或建议，请发邮件至：unity3d_api@163.com。

最后，祝愿图灵教育为 IT 行业贡献更多有价值的图书！

目　　录

第 1 章

Application类



Application类不含实例属性和实例方法，在脚本中通过直接调用Application类的静态属性和静态方法来控制程序的运行时数据，如场景的管理、数据的加载等。本章主要介绍Application类的一些静态属性和静态方法。

1.1 Application 类静态属性

在Application类中，涉及的静态属性主要有dataPath和loadedLevel。由于Application类的persistentDataPath属性、streamingAssetsPath属性和temporaryCachePath属性的功能与dataPath属性功能相近，因此把这些属性放到一起介绍。下面详细介绍这些属性。

1.1.1 dataPath属性：数据文件路径

基本语法 public static string dataPath { get; }

功能说明 此属性用于返回程序的数据文件所在文件夹的路径（只读）。返回路径为相对路径，不同游戏平台的数据文件保存路径不同，具体如下所示。

- Unity Editor：<工程文件夹所在路径>/Assets。
- Mac player：<应用程序路径>/Contents。
- iPhone player：<应用程序路径>/<AppName.app>/Data。
- Win player：<包含可执行文件的文件夹路径>/Data。
- Web player：播放器数据文件夹的绝对路径（没有实际的数据文件名称）。
- Flash：播放器数据文件夹的绝对路径（没有实际的数据文件名称）。

提　示 与此属性功能相近的属性有persistentDataPath、streamingAssetsPath和temporary-CachePath，它们的具体功能如下所示。

- persistentDataPath：此属性用于返回一个持久化数据存储目录的路径（只读），可以在此路径下存储一些持久化的数据文件。对于同一平台，在不同程序中调用此属性时，其返回值是相同的，但是在不同的运行平台下，其返回值会不一样。

- ❑ streamingAssetsPath：此属性用于返回流数据的缓存目录，返回路径为相对路径，适合设置一些外部数据文件的路径。
- ❑ temporaryCachePath：此属性用于返回一个临时数据的缓存目录（只读）。对于同一平台，在不同程序中调用此属性时，其返回值是相同的，但是在不同的运行平台下，其返回值是不一样的。

实例演示　以下代码演示了4种不同路径的属性输出值：

```
using UnityEngine;
using System.Collections;

public class DataPath_ts : MonoBehaviour
{
    void Start()
    {
        //4种不同的路径，都为只读
        //dataPath和streamingAssetsPath的路径位置一般是相对程序的安装目录位置
        //persistentDataPath和temporaryCachePath的路径位置一般是相对所在系统的固定位置
        Debug.Log("dataPath:" + Application.dataPath);
        Debug.Log("persistentDataPath:" + Application.persistentDataPath);
        Debug.Log("streamingAssetsPath:" + Application.streamingAssetsPath);
        Debug.Log("temporaryCachePath:" + Application.temporaryCachePath);
    }
}
```

在这段代码中，分别打印出了4种不同的路径模式，其中dataPath和streamingAssetsPath这两个属性的返回值一般是相对于程序安装目录的位置。由于是相对路径，这两个属性非常适用于在多平台移植中设置要读取的外部数据文件的路径。在第15章的综合实例中，就用到了dataPath这个属性。而persistentDataPath和temporaryCachePath这两个属性的返回值一般是程序所在平台的固定位置，对于不同的平台，其位置是不一样的，适合存放程序运行过程中产生的一些数据文件。图1-1是程序在我电脑中的运行输出，从输出结果可以看出这4种属性的不同路径返回值。

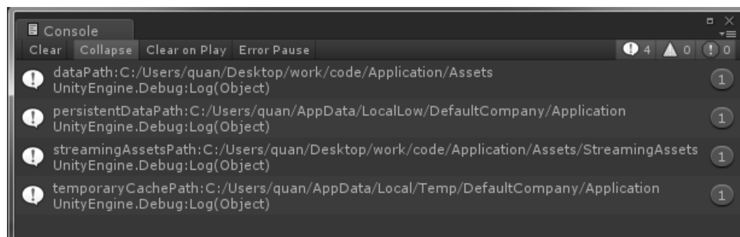


图1-1　dataPath实例演示的运行结果

1.1.2　loadedLevel属性：关卡索引

基本语法　`public static int loadedLevel { get; }`

功能说明 此属性用于返回当前程序最后加载的关卡（即Scene）的索引值（只读）。

实例演示 下面通过实例演示loadedLevel属性的使用方法。

```
using UnityEngine;
using System.Collections;
public class LoadedLevel_ts : MonoBehaviour
{
    void Start()
    {
        //返回当前场景的索引值
        Debug.Log("loadedLevel:" + Application.loadedLevel);
        //返回当前场景的名字
        Debug.Log("loadedLevelName:" + Application.loadedLevelName);
        //是否有场景正在被加载
        //在使用Application类的静态方法LoadLevel或LoadLevelAdditive加载一个新的场景时，
        //常常需要持续一段时间才能加载完毕，当场景加载完毕时，isLoadingLevel返回true，
        //否则返回false
        Debug.Log("isLoadingLevel:" + Application.isLoadingLevel);
        //返回游戏中可被加载的场景数量
        Debug.Log("levelCount:" + Application.levelCount);
        //返回当前游戏的运行平台
        //游戏的运行平台有很多种，例如手机、电脑、游戏机等，具体类型可在枚举类
        //RuntimePlatform中查看
        Debug.Log("platform:" + Application.platform);
        //当前游戏是否正在运行
        Debug.Log("isPlaying:" + Application.isPlaying);
        //当前游戏是否处于Unity编辑模式
        Debug.Log("isEditor:" + Application.isEditor);
    }
}
```

在这段代码中，分别打印出了Application类的一些常用静态属性，具体的属性功能请参考代码中的注释。图1-2是代码执行的输出结果，其中levelCount的值为2，这是因为在BuildSettings（File→BuildSettings）中有两个场景，具体请到工程中查看。

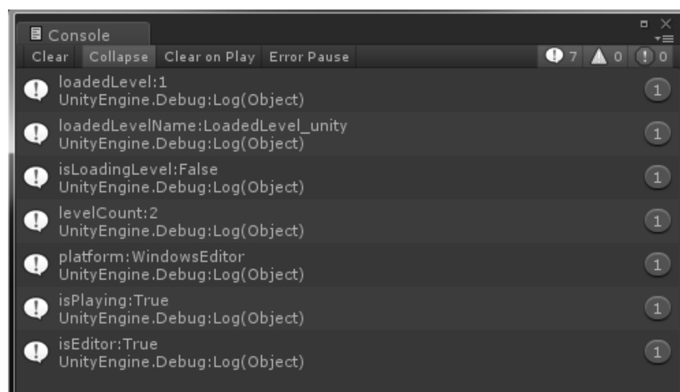


图1-2 属性loadedLevel的实例演示的运行结果

1.2 Application 类静态方法

在Application类中，涉及的静态方法有CaptureScreenshot方法、LoadLevelAdditiveAsync方法和RegisterLogCallback方法，下面详细介绍这些方法。

1.2.1 CaptureScreenshot方法：截屏

基本语法 (1) public static void CaptureScreenshot(string filename);

(2) public static void CaptureScreenshot(string filename, int superSize);

其中参数filename为截屏文件名称，superSize为放大系数，默认为0，即不放大。

功能说明 此方法用于截取当前游戏画面并将截取的图片保存为PNG格式。截屏后文件会默认保存在根目录下，如果根目录下已存在同名文件，将会被替换。当superSize大于1时，截屏文件的宽度和高度将同时被放大superSize倍。

提　　示
- 此方法在 Web 模式下无效。
- 当放大系数小于 0 时，按默认值 0 处理，即图片不放大也不缩小。

实例演示 下面通过实例演示如何使用CaptureScreenshot方法来截屏。

```
using UnityEngine;
using System.Collections;

public class CaptureScreenshot_ts : MonoBehaviour
{
    int tp = -1;
    void Update()
    {
        if (tp == 0)
        {
            //默认值，不放大
            Application.CaptureScreenshot("test01.png", 0);
        }
        else if (tp == 1)
        {
            //放大系数为1，即不放大
            Application.CaptureScreenshot("test02.png", 1);
        }
        else
        {
            //放大系数为2，即放大2倍
            Application.CaptureScreenshot("test03.png", 2);
        }
        tp++;
    }
}
```

在这段代码中，首先声明了一个int类型的变量tp，然后在Update方法中根据不同的变量值生成了3张不同放大系数的PNG图片。请自行运行程序，在程序根目录下查看生成的PNG文件的大小。由于CaptureScreenshot方法在每帧中只执行一次，所以如果使用如下代码的话，只会生成一张PNG图片，即test03.png：

```
using UnityEngine;
using System.Collections;

public class CaptureScreenshot_ts : MonoBehaviour
{
    void Update() {
        Application.CaptureScreenshot("test01.png", 0);
        Application.CaptureScreenshot("test02.png", 1);
        Application.CaptureScreenshot("test03.png", 2);
    }
}
```

在Update方法中，依次调用了3次CaptureScreenshot方法，试图生成3个不同放大系统的图片，但是在同一帧中，只有最后一次调用才能生效，故此段程序的运行结果只能生成一张PNG图片，即test03.png。

由于CaptureScreenshot方法可以实时截取程序屏幕，因此可以用其截图所包含的信息做一些有趣的应用，下面是一个小例子。

在本例中，玩家可以在程序左上角的TextField输入框里输入几个文字（最多不要超过输入框的最大宽度），单击“确定”按钮便会看到很多小球组成了文字的形状分布在三维空间中。图1-3是程序启动后的截图，图1-4是单击“确定”后的截图，图1-5是在文本框中输入“霞光满天”四个字后显示的形状。

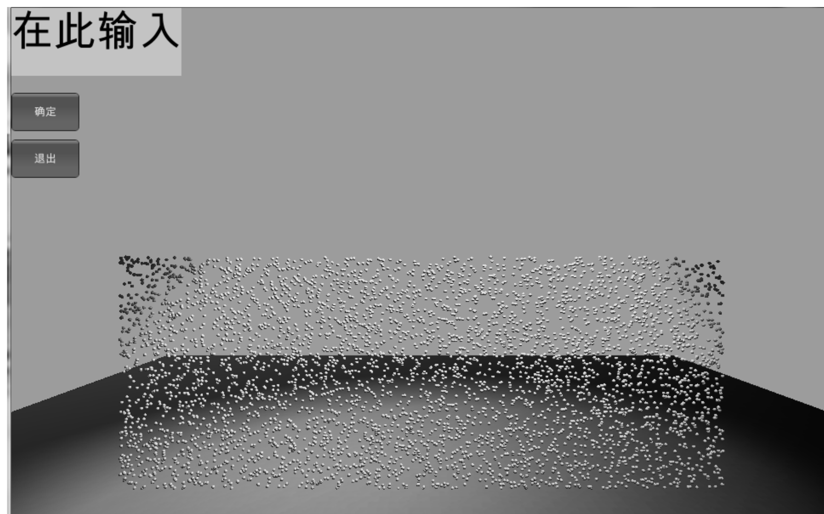


图1-3 程序启动后界面



图1-4　单击“确定”按钮后的界面

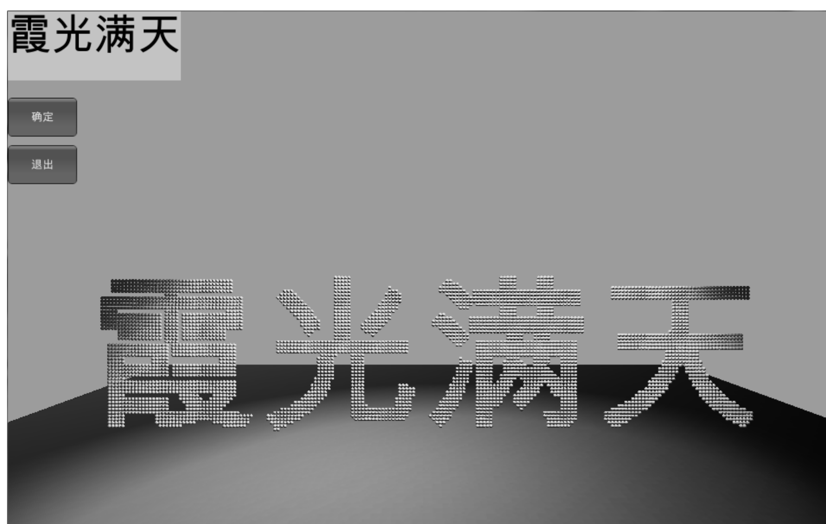


图1-5　在输入框中输入“霞光满天”后的效果

本例中用到的脚本有Capture_Use_ts.cs和Capture_Use_Sub_ts.cs，其中脚本Capture_Use_Sub_ts.cs用来控制每个小球的位置，其代码如下。

```
using UnityEngine;
using System.Collections;

public class Capture_Use_Sub_ts : MonoBehaviour
{
    //物体移动的目标位置
```

```
    public Vector3 toward;
    //物体移动时间
    float delays;
    void Start()
    {
        //获取一个随机值
        delays = Random.Range(2.0f, 4.0f);
    }

    void Update()
    {
        //通过更改物体位置来达到物体运动的效果
        transform.position = Vector3.MoveTowards(transform.position, toward, delays);
    }
}
```

在这个脚本中，首先声明了一个公共变量toward，以便在主脚本Capture_Use_ts.cs中调用，然后给另一个变量delays赋予了一个随机值，以便每个小球的运动显得更加真实。具体每行代码的作用已作了注释，不再赘述。

下面是主脚本Capture_Use_ts.cs中的代码。

```
using UnityEngine;
using System.Collections;

public class Capture_Use_ts : MonoBehaviour
{
    //td：用来指向屏幕截图
    Texture2D td = null;
    //txt_bg：用来指向文本输入的背景图片
    //可以指向一张纯色的图片，也可以自定义一个纯色的背景
    //本程序自定义了一个纯色的背景
    Texture2D txt_bg;
    //txt_w和txt_h用来记录文本框的宽度和高度
    int txt_w, txt_h;
    //my_save_path：用来记录截图的保存路径
    string my_save_path = "";
    //show_txt：用来记录输入的文本
    string show_txt = "在此输入";
    //my_colors：用来记录文本输入框的纹理颜色
    Color[] my_colors;
    //_w和_h用来记录my_colors的宽度和高度
    int _w, _h;
    //step：用来记录当前状态，step共有4种状态：
    //step=0时，是等待状态，等待在文本框中输入文本
    //step=1时，即点击“确定”按钮后，生成截图并保存
    //step=2时，读取截图信息
    //step=3时，是对读取的截图信息进行有选择的提取，并生成想要展示的内容
    int step = 0;
    //go：用来指向拼字所用物体对象
    public GameObject go;
    //gos：用来记录所有的go对象
    GameObject[] gos;
```

```
//gos_max用来记录gos的最大容量
int gos_max;
//gos_cur用来记录gos当前使用值
int gos_cur = 0;
//is_show: 用来记录图像矩阵中某个点是否需要展示物体
bool[,] is_show;

void Start()
{
    //初始化文本框的宽度和高度
    txt_w = 200;
    txt_h = 80;
    //初始化截取区间的大小，为了避免边界识别错误，其值应该比文本框的宽度和高度少几个像素
    _w = txt_w;
    _h = txt_h;
    _w -= 5;
    _h -= 5;
    //自定义txt_bg纹理
    txt_bg = new Texture2D(txt_w, txt_h);
    Color[] tdc = new Color[txt_w * txt_h];
    for (int i = 0; i < txt_w * txt_h; i++)
    {
        //所用像素点颜色应相同
        tdc[i] = Color.white;
    }
    txt_bg.SetPixels(0, 0, txt_w, txt_h, tdc);

    is_show = new bool[_h, _w];
    //初始化gos_max,其值大小为_w * _h的三分之一在一般情况下就够用了
    gos_max = _w * _h / 3;
    //实例化gos，使其随机分布_w和_h的区间内
    gos = new GameObject[gos_max];
    for (int i = 0; i < gos_max; i++)
    {
        gos[i] = (GameObject)Instantiate(go, new Vector3(Random.value * _w, Random.value *
            h, 10.0f), Quaternion.identity);
        gos[i].GetComponent<Capture_Use_Sub_ts>().toward = gos[i].transform.position;
    }
    //存储初始界面截图
    my_save_path = Application.persistentDataPath;
    Application.CaptureScreenshot(my_save_path + "/ts02.png");

}

void Update()
{
    switch (step)
    {
        case 0:
            break;
        case 1:
            step = 0;
            //截图并保存
            my_save_path = Application.persistentDataPath;
```

```
            Application.CaptureScreenshot(my_save_path + "/ts02.png");
            //给电脑一点儿时间用来保存新截取的图片
            Invoke("My_WaitForSeconds", 0.4f);
            Debug.Log(my_save_path);
            break;
        case 2:
            //由于在读取截图纹理的时候，一帧之内可能无法读完
            //所以需要step=0，避免逻辑出现混乱
            step = 0;
            //读取截图信息
            my_save_path = Application.persistentDataPath;
            StartCoroutine(WaitLoad(my_save_path + "/ts02.png"));
            break;
        case 3:
            //在计算并生成展示信息的时候，一帧之内可能无法完成
            //所以需要step=0，避免逻辑出现混乱
            step = 0;
            //计算并生成展示信息
            Cum();
            break;
    }
}

//计算并生成展示信息
void Cum()
{
    if (td != null)
    {
        float ft;
        //ft：用来记录文本框左下角像素的R通道值，用来作为后面的参照
        ft = td.GetPixel(2, td.height - _h).r;
        //截取文本框纹理信息
        //需要注意的是，纹理坐标系和GUI坐标系不同
        //纹理坐标系以左下角为原点，而GUI坐标系以左上角为原点
        //以2为x方向起点是为了避免截图边缘的痕迹
        my_colors = td.GetPixels(2, td.height - _h, _w, _h);
        int l = my_colors.Length;
        Debug.Log("length: " + l);
        //通过遍历my_colors的R值，将其与ft比较来确定是否需要展示物体
        for (int i = 0; i < l; i++)
        {
            is_show[i / _w, i % _w] = my_colors[i].r == ft ? false : true;
        }
        //根据is_show的值排列gos中物体的位置
        for (int i = 0; i < _h; i++)
        {
            for (int j = 0; j < _w; j++)
            {
                if (is_show[i, j])
                {
                    if (gos_cur < gos_max)
                    {
                        gos[gos_cur].GetComponent<Capture_Use_Sub_ts>().toward = new
                            Vector3(j, i, 0.0f);
```

```
                        gos[gos_cur].SetActive(true);
                        gos_cur++;
                    }
                    //当前gos数量不够用时需要扩充gos的容量
                    else
                    {
                        Debug.Log("容量过小");
                        int temps = gos_max;
                        //将gos容量扩大1.5倍
                        gos_max = (int)(gos_max * 1.5f);
                        GameObject[] tps = new GameObject[gos_max];
                        for (int k = 0; k < temps; k++)
                        {
                            tps[k] = gos[k];
                        }
                        for (int k = temps; k < gos_max; k++)
                        {
                            tps[k] = (GameObject)Instantiate(go, new Vector3(Random.value *
                                h, Random.value * _w, 10.0f), Quaternion.identity);
                            tps[k].GetComponent<Capture_Use_Sub_ts>().toward = tps[k].
                                transform.position;
                        }

                        gos = new GameObject[gos_max];
                        gos = tps;

                        gos[gos_cur].GetComponent<Capture_Use_Sub_ts>().toward = new
                            Vector3(j, i, 0.0f);
                        gos[gos_cur].SetActive(true);
                        gos_cur++;

                    }

                }
            }
        }
        //隐藏gos中未曾用到的物体
        for (int k = gos_cur; k < gos_max; k++)
        {
            gos[k].SetActive(false);
        }
    }
}
//绘制界面
void OnGUI()
{
    //绘制纹理作为TextField的背景
    GUI.DrawTexture(new Rect(0.0f, 0.0f, txt_w, txt_h), txt_bg);
    GUIStyle gs = new GUIStyle();
    gs.fontSize = 50;
    show_txt = GUI.TextField(new Rect(0.0f, 0.0f, txt_w, txt_h), show_txt, 15, gs);

    if (GUI.Button(new Rect(0, 100, 80, 45), "确定"))
    {
```

```
            //取消在TextField中的焦点
            GUI.FocusControl(null);
            //重置gos_cur的值
            gos_cur = 0;
            step = 1;
        }
        //程序退出
        if (GUI.Button(new Rect(0, 155, 80, 45), "退出"))
        {
            Application.Quit();
        }
    }

    //加载图片
    IEnumerator WaitLoad(string fileName)
    {
        WWW wwwTexture = new WWW("file://" + fileName);
        Debug.Log(wwwTexture.url);
        yield return wwwTexture;
        td = wwwTexture.texture;
        step = 3;
    }
    //进入步骤2
    void My_WaitForSeconds()
    {
        step = 2;
    }
}
```

此脚本用来生成截图、读取截图信息以及排列小球的位置，具体过程已在代码中做了详细注释，在此不再赘述，更多信息请到工程中查看。

1.2.2　LoadLevelAdditiveAsync方法：异步加载关卡

基本语法　(1) `public static AsyncOperation LoadLevelAdditiveAsync(int index);`

其中参数index是被加载关卡的索引值，可以在菜单File→Build Settings中查看关卡的索引值。

(2) `public static AsyncOperation LoadLevelAdditiveAsync(string levelName);`

其中参数levelName是被加载关卡的名字，可以在菜单File→Build Settings中查看关卡的名字。

功能说明　此方法用于按照关卡名字在后台异步加载关卡到当前场景中，此方法只是将新关卡加载到当前场景，当前场景的原有内容不会被销毁。此方法仅专业版可用。

实例演示　以下代码是第15章中Game01.cs脚本中的代码片段。

```
IEnumerator Start()
    {
```

```
        AsyncOperation async = Application.LoadLevelAdditiveAsync("Game02");
        //异步加载中
        Debug.Log("1:" + async.isDone);//是否加载完成
        Debug.Log("2:" + async.progress);//加载进度，范围0～1
        yield return async;
        //加载完成后
        Debug.Log("3:" + async.isDone);
        Debug.Log("4:" + async.progress);
        is_load_over = async.isDone;
    }
```

在这段代码中，首先在Start方法前加IEnumerator标示，接着声明了一个异步操作async并开始加载一个名为“Game02”的场景，当加载完成后，属性async.isDone的值会返回true。具体运行情况请结合第15章查看。

1.2.3 RegisterLogCallback 方法：注册委托

基本语法 public static void RegisterLogCallback(Application.LogCallback handler);

其中参数handler是委托方法的名字。

功能说明 此方法用于注册一个委托来调用日志信息。

提　　示 RegisterLogCallbackThreaded方法与此方法的功能相似，不同之处在于RegisterLogCallbackThreaded方法是在一个新的线程中调用委托。

实例演示 下面通过实例演示如何注册委托并输出日志信息。

```
using UnityEngine;
using System.Collections;

public class RegisterLogCallback_ts : MonoBehaviour
{
    string output = "";//日志输出信息
    string stack = "";//堆栈跟踪信息
    string logType = "";//日志类型
    int tp = 0;
  //打印日志信息
    void Update()
    {
        Debug.Log("output:" + output);
        Debug.Log("stack:" + stack);
        Debug.Log("logType:" + logType);
        Debug.Log("tp:"+(tp++));
    }
    void OnEnable()
    {
        //注册委托
        Application.RegisterLogCallback(MyCallback);
    }
    void OnDisable()
```

```
    {
        //取消委托
        Application.RegisterLogCallback(null);
    }
    //委托方法
    //方法名字可以自定义，但方法的参数类型要符合Application.LogCallback中的参数类型
    void MyCallback(string logString, string stackTrace, LogType type)
    {
        output = logString;
        stack = stackTrace;
        logType = type.ToString();
    }
}
```

在这段代码中，首先声明了3个变量output、stack和logType，分别用来存储日志的输出信息、堆栈跟踪信息和日志类型。然后在方法OnEnable中注册一个名为MyCallback的委托方法。最后在Update方法中输出日志信息，图1-6是一张运行时截图。使用此方法可以通过output值来获取日志的输出信息，通过stack值来追踪output的输出位置。

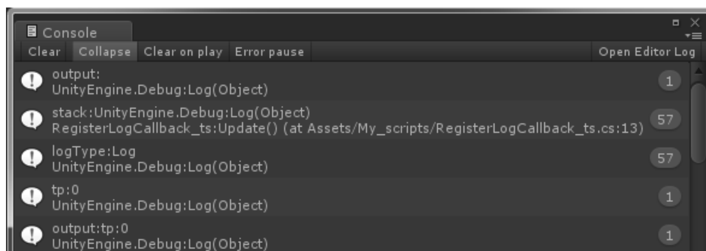


图1-6　RegisterLogCallback实例演示的运行结果

第 2 章

Camera类

2

Camera类用来控制游戏中虚拟场景的展示，以左下角为屏幕的(0,0)点坐标，以右上角为屏幕的(camera.pixelWidth，camera.pixelHeight)点坐标。如果用单位化方式表示，则左下角为(0,0)点，右上角为(1,1)点。本章主要介绍了Camera类的一些实例属性和实例方法，最后对摄像机的视口与aspect、pixelRect及rect之间的关系进行了注解。

2.1 Camera 类实例属性

在Camera类中，涉及的实例属性有aspect、cameraToWorldMatrix、cullingMask、eventMask、layerCullDistances、layerCullSpherical、orthographic、pixelRect、projectionMatrix、rect、renderingPath、targetTexture和worldToCameraMatrix，下面详细介绍这些属性。

2.1.1 aspect属性：设置摄像机视口比例

基本语法 public float aspect { get; set; }

功能说明 此属性用于获取或设置Camera视口的宽高比例值。例如，设camera.aspect=2.0f，则camera视口的宽度/高度=2.0f，但是当硬件显示器屏幕的宽度与高度比例不为2.0f时，视图的显示将会发生变形。aspect只处理摄像机Camera可以看到的视图的宽高比例，而硬件显示屏的作用只是把摄像机Camera看到的内容显示出来，当硬件显示屏的宽高比例与aspect的比例值不同时，视图将发生变形。关于此属性的更多内容请参考2.3节。

实例演示 下面通过实例演示不同的aspect值对Camera视口的影响。

```
using UnityEngine;
using System.Collections;

public class Aspect_ts : MonoBehaviour
{
    void Start()
    {
        //camera.aspect的默认值即为当前硬件的aspect值
```

```
        Debug.Log("camera.aspect的默认值: " + camera.aspect);
    }
    void OnGUI()
    {
        if (GUI.Button(new Rect(10.0f, 10.0f, 200.0f, 45.0f), "aspect=1.0f"))
        {
        //重置当前摄像机的aspect值
            camera.ResetAspect();
            camera.aspect = 1.0f;
        }
        if (GUI.Button(new Rect(10.0f, 60.0f, 200.0f, 45.0f), "aspect=2.0f"))
        {
        //重置当前摄像机的aspect值
            camera.ResetAspect();
            camera.aspect = 2.0f;
        }
        if (GUI.Button(new Rect(10.0f, 110.0f, 200.0f, 45.0f), "aspect还原默认值"))
        {
        //重置当前摄像机的aspect值
            camera.ResetAspect();
        }
    }
}
```

在这段代码中，首先在Start方法中打印出了Camera的默认aspect值，camera.aspect的默认值即为当前硬件的aspect值，若在Unity编辑模式中运行即为Game视图中aspect的值，如图2-1所示。然后在OnGUI方法中定义了3个Button来切换不同的aspect值，在每次设置新的aspect值之前，都需要先调用ResetAspect方法来重置视口的aspect值。具体的变化情况请自行运行程序查看。



图2-1 Game视图中的aspect值

2.1.2 cameraToWorldMatrix属性：变换矩阵

基本语法 public Matrix4x4 cameraToWorldMatrix { get; }

功能说明 此属性的功能是返回从摄像机的局部坐标系到世界坐标系的变换矩阵（只读）。

提　　示 Camera中的forward方向为其自身坐标系的–z轴方向，一般其他GameObject对象的forward方向为自身坐标系的z轴方向。

实例演示 下面通过实例演示属性cameraToWorldMatrix的使用。

```
using UnityEngine;
using System.Collections;

public class CameraToWorldMatrix_ts : MonoBehaviour
{
    void Start()
    {
        Debug.Log("Camera旋转前位置：" + transform.position);
        Matrix4x4 m = camera.cameraToWorldMatrix;
        //v3的值为沿着Camera局部坐标系的-z轴方向前移5个单位的位置在世界坐标系中的位置
        Vector3 v3 = m.MultiplyPoint(Vector3.forward * 5.0f);
        //v4的值为沿着Camera世界坐标系的-z轴方向前移5个单位的位置在世界坐标系中的位置
        Vector3 v4 = m.MultiplyPoint(transform.forward * 5.0f);
        //打印v3、v4
        Debug.Log("旋转前，v3坐标值：" + v3);
        Debug.Log("旋转前，v4坐标值：" + v4);
        transform.Rotate(Vector3.up * 90.0f);
        Debug.Log("Camera旋转后位置：" + transform.position);
        m = camera.cameraToWorldMatrix;
        //v3的值为沿着Camera局部坐标系的-z轴方向前移5个单位的位置在世界坐标系中的位置
        v3 = m.MultiplyPoint(Vector3.forward * 5.0f);
        //v3的值为沿着Camera世界坐标系的-z轴方向前移5个单位的位置在世界坐标系中的位置
        v4 = m.MultiplyPoint(transform.forward * 5.0f);
        //打印v3、v4
        Debug.Log("旋转后，v3坐标值：" + v3);
        Debug.Log("旋转后，v4坐标值：" + v4);
    }
}
```

在这段代码中，首先打印出了Camera旋转前的位置，然后将摄像机的cameraToWorldMatrix值赋给Matrix4x4类变量m，然后使用Matrix4x4类的方法MultiplyPoint，计算出了摄像机分别沿着的局部坐标系的–z轴方向和沿着世界坐标系的–z轴方向前进5个像素的坐标位置，接着将摄像机沿着y轴正向旋转90度（此时摄像机的局部坐标系的z轴方向和世界坐标系的x轴方向一致），然后再使用Matrix4x4类的方法MultiplyPoint，计算出了摄像机分别沿着的局部坐标系的–z轴方向和沿着世界坐标系的–z轴方向前进5个像素的坐标位置，最后打印出了旋转后的v3和v4的值。程序运行结果如图2-2所示。MultiplyPoint的使用方法请参考本书Matrix4x4类中MultiplyPoint方法的功能说明。

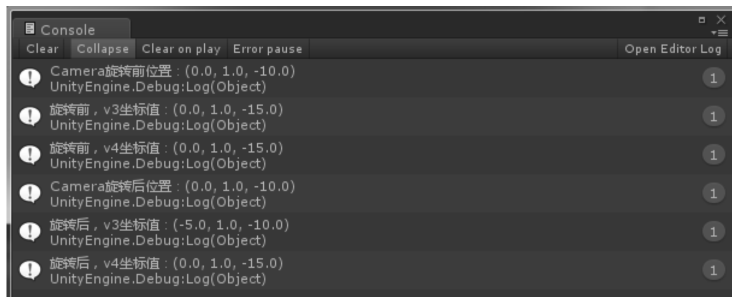


图2-2　cameraToWorldMatrix实例演示的运行结果

2.1.3 cullingMask属性：摄像机按层渲染

基本语法 public int cullingMask { get; set; }

功能说明 此属性用于按层（即GameObject.layer）有选择性地渲染场景中的物体。通过cullingMask可以使得当前摄像机有选择性地渲染场景中的部分物体，默认cullingMask=-1即渲染场景中任何层物体，当cullingMask=0时不渲染场景中任何层。若只渲染分别位于2、3、4层的物体，则可以使用代码cullingMask=(1<<2)+ (1<<3)+ (1<<4)来实现。

实例演示 下面通过实例演示属性cullingMask的使用。

```
using UnityEngine;
using System.Collections;

public class CullingMask_ts : MonoBehaviour
{
    void OnGUI()
    {
        //默认CullingMask=-1，即渲染任何层
        if (GUI.Button(new Rect(10.0f, 10.0f, 200.0f, 45.0f), "CullingMask=-1"))
        {
            camera.cullingMask = -1;
        }
        //不渲染任何层
        if (GUI.Button(new Rect(10.0f, 60.0f, 200.0f, 45.0f), "CullingMask=0"))
        {
            camera.cullingMask = 0;
        }
        //仅渲染第0层
        if (GUI.Button(new Rect(10.0f, 110.0f, 200.0f, 45.0f), "CullingMask=1<<0"))
        {
            camera.cullingMask = 1 << 0;
        }
        //仅渲染第8层
        if (GUI.Button(new Rect(10.0f, 160.0f, 200.0f, 45.0f), "CullingMask=1<<8"))
        {
            camera.cullingMask = 1 << 8;
        }
        //渲染第8层与第0层
        if (GUI.Button(new Rect(10.0f, 210.0f, 200.0f, 45.0f), "CullingMask=0&&8"))
        {
            //注：不可大意写成camera.cullingMask = 1 << 8+1;或
            //camera.cullingMask = 1+1<<8 ;因为根据运算符优先次序，其分别等价于
            //camera.cullingMask = 1 << (8+1)和camera.cullingMask = (1+1)<<8
            camera.cullingMask = (1 << 8) + 1;
        }
    }
}
```

在这段代码中，在OnGUI方法中定义了5个不同的Button来渲染不同层的物体。当然，在使用cullingMask来有选择性地渲染物体之前，需要先对场景中物体的层次进行设

置。另外需要注意渲染多个层时的代码写法，如以上代码所示，在渲染第8层和第0层时，请勿将代码写成camera.cullingMask = 1 << 8+1或camera.cullingMask = 1+1<<8的形式。具体的渲染情况请自行运行程序查看。

2.1.4 eventMask属性：按层响应事件

基本语法 public int eventMask { get; set; }

功能说明 此属性的功能是选择哪个层（layer）的物体可以响应鼠标事件，其使用说明如下。

- 如果要使物体响应鼠标事件必须首先满足如下两个条件。

  第一，物体在摄像机的视野范围内。

  第二，在2的layer次方的值与eventMask进行与运算（&）后结果为仍为2的layer次方的值，例如当前物体的层为Default，即layer值为0，则2的0次方为1，如果1与eventMask进行与运算后结果仍为1，则此物体便会响应鼠标事件。由于当eventMask为奇数时，与1的与运算结果都为1，所以若物体的层为Default并且eventMask为奇数时物体便会响应鼠标事件。
- 如果想要多个不同层的物体都响应鼠标事件，则需要把所有层的2的layer次方值相加，再与eventMask做与运算。例如，现有两个物体，它们的layer值分别为1和3，则当eventMask与9（因为$2^0+2^3=9$）进行与运算后若结果仍为9，则这两个物体都会响应鼠标事件。
- 此属性有一个特殊情况，当物体的layer选择IgnoreRaycast（其为系统内置，值为2）时，无论eventMask值为多少，物体都无法响应鼠标事件。原因很简单，因为这个层的物体都忽略了射线碰撞，鼠标就探测不到物体的存在，因而也就无法响应鼠标事件了。

提　示 此属性的计算比较消耗资源，在不使用鼠标事件时建议将值设为零。

实例演示 下面通过实例演示属性eventMask的使用。

```
using UnityEngine;
using System.Collections;

public class EventMask_ts : MonoBehaviour
{
    bool is_rotate = false;//控制物体旋转
    public Camera c;//指向场景中的摄像机
    //记录摄像机的eventMask值，可以在程序运行时在Inspector面板中修改值的大小
    public int eventMask_now = -1;
    //记录当前物体的层
    int layer_now;
    int tp;//记录2的layer次方的值
    int ad;//记录与运算（&）的结果
```

```
    string str = null;

    void Update()
    {
        //记录当前对象的层，可以在程序运行时在Inspector面板中选择不同的层
        layer_now = gameObject.layer;
        //求2的layer_now次方的值
        tp = (int)Mathf.Pow(2.0f, layer_now);
        //与运算（&）
        ad = eventMask_now & tp;
        c.eventMask = eventMask_now;
        //当is_rotate为true时旋转物体
        if (is_rotate)
        {
            transform.Rotate(Vector3.up * 15.0f * Time.deltaTime);
        }
    }
    //当鼠标左键按下时，物体开始旋转
    void OnMouseDown()
    {
        is_rotate = true;
    }
    //当鼠标左键抬起时，物体结束旋转
    void OnMouseUp()
    {
        is_rotate = false;
    }
    void OnGUI()
    {
        GUI.Label(new Rect(10.0f, 10.0f, 300.0f, 45.0f), "当前对象的layer值为：" + layer_now
            + " ，2的layer次方的值为" + tp);
        GUI.Label(new Rect(10.0f, 60.0f, 300.0f, 45.0f), "当前摄像机eventMask的值为：" +
            eventMask_now);
        GUI.Label(new Rect(10.0f, 110.0f, 500.0f, 45.0f), "根据算法，当eventMask的值与" + tp
            + "进行与运算（&）后， 若结果为" + tp + "，则物体响应OnMousexxx方法，否则不响应！
            ");

        if (ad == tp)
        {
            str = " ，所以物体会响应OnMouseXXX方法！";
        }
        else
        {
            str = " ，所以物体不会响应OnMouseXXX方法！";
        }
        GUI.Label(new Rect(10.0f, 160.0f, 500.0f, 45.0f), "而当前eventMask与" + tp + "进行与
            运算（&）的结果为" + ad + str);
    }
}
```

这段代码已对各段代码的功能做了较为详细的注释，请自行运行程序查看。在运行程序后，请在脚本所在物体的Inspector面板中选择不同的layer并设置不同的eventMask参数，然后查看界面文字的说明，并用鼠标点击相应的物体，查看物体的变化状态。

图2-3是一张程序运行时截图。



图2-3　eventMask实例演示运行截图

2.1.5　layerCullDistances属性：层消隐的距离

基本语法　public float[] layerCullDistances { get; set; }

功能说明　此属性用来设置摄像机基于层的消隐距离。摄像机可以通过基于层（GameObject.layer）的方式来设置不同层物体的消隐距离，但这个距离必须小于或等于摄像机的farClipPlane才有效。

实例演示　下面通过实例演示属性layerCullDistances的使用。

```
using UnityEngine;
using System.Collections;
public class LayerCullDistances_ts : MonoBehaviour
{
    public Transform cb1;
    void Start()
    {
        //定义大小为32的一维数组，用来存储所有层的剔除距离
        float[] distances = new float[32];
        //设置第9层的剔除距离
        distances[8] = Vector3.Distance(transform.position, cb1.position);
        //将数组赋给摄像机的layerCullDistances
        camera.layerCullDistances = distances;
    }
    void Update()
    {
        //摄像机远离物体
        transform.Translate(transform.right * Time.deltaTime);
    }
}
```

在这段代码中，首先在Start方法中定义了一个大小为32的一维数组distance (Unity共可设置32层)用来存储各层的剔除距离，然后对第9层即distance[8]的大小进行赋值，接着将distance赋给摄像机的layerCullDistances，最后在Undate方法中使用Translate方法使摄像机慢慢远离物体。运行程序可以发现，当物体cube1的所有面与Camera的距离都大于剔除距离时，cube1将会消失，而cube2依然可见。当然，在使用layerCullDistances功能时，需要先对场景中各个物体所在的层进行设置，例如设置物体Cube1的层，如图2-4所示。



图2-4 设置物体Cube1的层

2.1.6 layerCullSpherical属性：基于球面距离剔除

基本语法 public bool layerCullSpherical { get; set; }

功能说明 此属性用于设置摄像机在基于层剔除物体时，是否采用基于球面距离的剔除方式。此属性默认值为false，即不使用球面剔除方式，此时，只要物体表面上有一点没有超出物体所在层的远视口平面，物体就是可见的。当layerCullSpherical为true时，只要物体的世界坐标点position与摄像机的距离大于所在层的剔除距离，物体就是不可见的，这和值为false时的计算方式不同。

实例演示 下面通过实例演示属性layerCullSpherical的使用。在本实例中，创建了3个Cube对象，其中Cube1在摄像机的正前方，Cube2和Cube3在Cube1的两侧，并将如下脚本赋给MainCamera，具体请查看工程中的设置。

```
using UnityEngine;
using System.Collections;

public class layerCullSpherical_ts : MonoBehaviour
{
    public Transform cb1, cb2, cb3;
    void Start()
    {
        //定义大小为32的一维数组，用来存储所有层的剔除距离
        float[] distances = new float[32];
        //设置第9层的剔除距离
        distances[8] = Vector3.Distance(transform.position, cb1.position);
        //将数组赋给摄像机的layerCullDistances
        camera.layerCullDistances = distances;
        //打印出3个物体距离摄像机的距离
        Debug.Log("Cube1距离摄像机的距离："+Vector3.Distance(transform.position, cb1.position));
```

```
        Debug.Log("Cube2距离摄像机的距离：" + Vector3.Distance(transform.position, cb2.position));
        Debug.Log("Cube3距离摄像机的距离：" + Vector3.Distance(transform.position, cb3.position));
    }

    void OnGUI()
    {
        //使用球形距离剔除
        if (GUI.Button(new Rect(10.0f, 10.0f, 180.0f, 45.0f), "use layerCullSpherical"))
        {
            camera.layerCullSpherical = true;
        }
        //取消球形距离剔除
        if (GUI.Button(new Rect(10.0f, 60.0f, 180.0f, 45.0f), "unuse layerCullSpherical"))
        {
            camera.layerCullSpherical = false;
        }
    }
}
```

在这段代码中，首先声明了3个Transform变量，用于指向场景中的物体，然后在Start方法中将Cube1与摄像机的距离作为场景中第9层的剔除距离，接着分别打印出3个物体距离摄像机的距离，最后在OnGUI方法中定义了两个Button，用于控制是否使用球形距离剔除方式。运行程序可以发现，当选择使用球形距离剔除时，物体Cube2和Cube3将不可见，并且当Cube1与摄像机的距离稍微增大一点时（例如将Cube1的z轴分量增加0.1f），Cube1也将不可见。当取消使用球形距离剔除时，3个物体均可见，并且只要物体表面上有一点与摄像机的距离没有超出剔除距离，此物体便是可见的，即在这种情况下，物体是否被剔除还和物体的scale值有关。请自行运行程序查看。

2.1.7　orthographic属性：摄像机投影模式

基本语法　public bool orthographic { get; set; }

功能说明　此属性用于获取或设置当前摄像机的投影模式，投影模式包括正交投影模式（orthographic）和透视投影模式（perspective）。若值为true则为正交投影，反之为透视投影。正交投影模式下，物体在视口中的大小只与正交视口的大小有关，与摄像机到物体的距离无关，主要用来呈现2D效果。而在透视投影模式下，物体在视口中的大小与摄像机的视口夹角（field of view）以及摄像机与物体的距离都有关系，有远小近大的效果，主要用来呈现3D效果。

实例演示　下面通过实例演示属性orthographic的使用。本实例中有两个大小一样的Cube对象，设置它们的位置，使得Cube1比Cube2在y轴上更接近摄像机。

```
using UnityEngine;
using System.Collections;

public class orthographic_ts : MonoBehaviour
{
```

```
        float len = 5.5f;
        void OnGUI()
        {
            if (GUI.Button(new Rect(10.0f, 10.0f, 120.0f, 45.0f), "正交投影"))
            {
                camera.orthographic = true;
                len = 5.5f;
            }
            if (GUI.Button(new Rect(150.0f, 10.0f, 120.0f, 45.0f), "透视投影"))
            {
                camera.orthographic = false;
                len = 60.0f;
            }
            if (camera.orthographic)
            {
                //正交投影模式下，物体没有远大近小的效果，
                //orthographicSize的大小无限制，当orthographicSize为负数时视口的内容会颠倒，
                //orthographicSize的绝对值为摄像机视口的高度值，即上下两条边之间的距离
                len = GUI.HorizontalSlider(new Rect(10.0f, 60.0f, 300.0f, 45.0f), len, -20.0f,
                    20.0f);
                camera.orthographicSize = len;
            }
            else
            {
                //透视投影模式下，物体有远大近小的效果
                //fieldOfView的取值范围为1.0-179.0
                len = GUI.HorizontalSlider(new Rect(10.0f, 60.0f, 300.0f, 45.0f), len, 1.0f,
                    179.0f);
                camera.fieldOfView = len;
            }
            //实时显示len大小
            GUI.Label(new Rect(320.0f, 60.0f, 120.0f, 45.0f), len.ToString());
        }
    }
```

在这段代码中，首先声明了一个变量len，用于记录Slider的值，然后在OnGUI方法中定义了两个按钮，用来设置摄像机的投影方式，最后根据不同投影方式下Slider的值来控制视口的大小（orthographicSize或fieldOfView），请自行运行程序查看不同投影方式下视口的变化。

2.1.8 pixelRect属性：摄像机渲染区间

基本语法 public Rect pixelRect { get; set; }

功能说明 此属性用于设置camera被渲染到屏幕中的坐标位置。pixelRect与属性rect功能类似，不同的是pixelRect以实际像素大小来设置显示视口的位置，而rect以单位化方式设置显示视口的位置。设camera.pixelRect的值为(x_0, y_0,w,h)，如图2-5所示，A为原始平面大小，B为变换后的视口大小，则x_0的值为视口右移的像素大小，y_0的值为视口上移的像素大小，w的值为camera.pixelWidth，h的值为camera.pixelHeight。

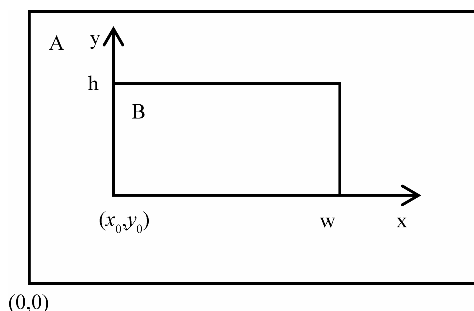


图2-5　pixelRect放缩示意图

另外，Screen.width和Screen.height为模拟硬件屏幕的宽高值，不随camera.pixelWidth和camera.pixelHeight的改变而改变。更多内容请参考2.3节。

实例演示　下面通过实例演示属性pixelRect的使用。

```
using UnityEngine;
using System.Collections;

public class PixelRect_ts : MonoBehaviour
{
    int which_change = -1;
    float temp_x = 0.0f, temp_y = 0.0f;
    void Update()
    {
        //Screen.width和Screen.height为模拟硬件屏幕的宽高值
        //其返回值不随camera.pixelWidth和camera.pixelHeight的改变而改变
        Debug.Log("Screen.width:" + Screen.width);
        Debug.Log("Screen.height:" + Screen.height);
        Debug.Log("pixelWidth:" + camera.pixelWidth);
        Debug.Log("pixelHeight:" + camera.pixelHeight);
        //通过改变Camera的坐标位置而改变视口的区间
        if (which_change == 0)
        {
            if (camera.pixelWidth > 1.0f)
            {
                temp_x += Time.deltaTime * 20.0f;
            }
            //取消以下注释查看平移状况
            //if (camera.pixelHeight > 1.0f)
            //{
            //    temp_y += Time.deltaTime * 20.0f;
            //}
            camera.pixelRect = new Rect(temp_x, temp_y, camera.pixelWidth,
                camera.pixelHeight);
        }
        //通过改变Camera的视口宽度和高度来改变视口的区间
        else if (which_change == 1)
        {
            if (camera.pixelWidth > 1.0f)
```

```
                {
                    temp_x = camera.pixelWidth - Time.deltaTime * 20.0f;
                }
                //取消以下注释查看平移状况
                //if (camera.pixelHeight > 1.0f)
                //{
                //    temp_y = camera.pixelHeight - Time.deltaTime * 20.0f
                //}
                camera.pixelRect = new Rect(0, 0, temp_x, temp_y);
            }
        }
        void OnGUI()
        {
            if (GUI.Button(new Rect(10.0f, 10.0f, 200.0f, 45.0f), "视口改变方式1"))
            {
                camera.rect = new Rect(0.0f, 0.0f, 1.0f, 1.0f);
                which_change = 0;
                temp_x = 0.0f;
                temp_y = 0.0f;
            }
            if (GUI.Button(new Rect(10.0f, 60.0f, 200.0f, 45.0f), "视口改变方式2"))
            {
                camera.rect = new Rect(0.0f, 0.0f, 1.0f, 1.0f);
                which_change = 1;
                temp_x = 0.0f;
                temp_y = camera.pixelHeight;
            }
            if (GUI.Button(new Rect(10.0f, 110.0f, 200.0f, 45.0f), "视口还原"))
            {
                camera.rect = new Rect(0.0f, 0.0f, 1.0f, 1.0f);
                which_change = -1;
            }
        }
    }
```

在这段代码中，首先定义了一个变量which_change，然后在OnGUI方法中定义了3个Button来更改which_change的值，最后在Update方法中根据不同的which_change值通过pixelRect属性来实现不同的视口变换。具体运行情况请自行运行程序查看。

2.1.9 projectionMatrix属性：自定义投影矩阵

基本语法 public Matrix4x4 projectionMatrix { get; set; }

功能说明 此属性的功能是设置摄像机的自定义投影矩阵。此属性常在一些特效场景下用到，在切换变换矩阵时通常需要先用camera.ResetProjectionMatrix()重置camera的变换矩阵。

实例演示 下面通过实例演示属性projectionMatrix的使用。

```
using UnityEngine;
using System.Collections;

public class ProjectionMatrix_ts : MonoBehaviour
```

```
{
    public Transform sp, cb;
    public Matrix4x4 originalProjection;
    float q=0.1f;//晃动振幅
    float p=1.5f;//晃动频率
    int which_change = -1;
    void Start()
    {
     //记录原始投影矩阵
     originalProjection = camera.projectionMatrix;
    }
    void Update()
    {
        sp.RotateAround(cb.position, cb.up, 45.0f * Time.deltaTime);
        Matrix4x4 pr = originalProjection;
        switch (which_change)
        {
            case -1:
                break;
            case 0:
                //绕摄像机x轴晃动
                pr.m11 += Mathf.Sin(Time.time * p) * q;
                break;
            case 1:
                //绕摄像机y轴晃动
                pr.m01 += Mathf.Sin(Time.time * p) * q;
                break;
            case 2:
                //绕摄像机z轴晃动
                pr.m10 += Mathf.Sin(Time.time * p) * q;
                break;
            case 3:
                //绕摄像机左右移动
                pr.m02 += Mathf.Sin(Time.time * p) * q;
                break;
            case 4:
                //摄像机视口放缩运动
                pr.m00 += Mathf.Sin(Time.time * p) * q;
                break;
        }
        //设置Camera的变换矩阵
        camera.projectionMatrix = pr;
    }
    void OnGUI()
    {
        if (GUI.Button(new Rect(10.0f, 10.0f, 200.0f, 45.0f), "绕摄像机x轴晃动"))
        {
            //重置摄像机的投影矩阵
            camera.ResetProjectionMatrix();
            which_change = 0;
        }
        if (GUI.Button(new Rect(10.0f, 60.0f, 200.0f, 45.0f), "绕摄像机y轴晃动"))
        {
            camera.ResetProjectionMatrix();
```

```
                which_change = 1;
            }
            if (GUI.Button(new Rect(10.0f, 110.0f, 200.0f, 45.0f), "绕摄像机z轴晃动"))
            {
                camera.ResetProjectionMatrix();
                which_change = 2;
            }
            if (GUI.Button(new Rect(10.0f, 160.0f, 200.0f, 45.0f), "绕摄像机左右移动"))
            {
                camera.ResetProjectionMatrix();
                which_change = 3;
            }
            if (GUI.Button(new Rect(10.0f, 210.0f, 200.0f, 45.0f), "视口放缩运动"))
            {
                camera.ResetProjectionMatrix();
                which_change = 4;
            }
        }
    }
```

在这段代码中，首先声明一个Matrix4x4变量originalProjection，然后在Start方法中将摄像机的原始投影矩阵赋给变量originalProjection，接着在Update方法中先将originalProjection值赋给一个新的变量pr,再根据不同的which_change对矩阵pr进行不同的变换，最后将pr赋给Camera的投影矩阵从而实现不同的效果。需要注意的是，在每次选择不同的效果之前需要调用方法ResetProjectionMatrix重置摄像机的投影矩阵。具体运行情况请自行运行程序查看。

2.1.10 rect属性：摄像机视图的位置和大小

基本语法 `public Rect rect { get; set; }`

功能说明 此属性使用单位化坐标系的方式来设置当前摄像机的视图位置和大小。如图2-6所示，A为原始视口大小，B为移位及放缩后的视口位置和大小。设camera.rect= x_0,y_0,w_0,h_0，则

- x_0的值为A物体向右移动了A的总宽度的w_1倍，例如x_0=0.1f则表示A向右移动的距离为$x_1=0.1*w_1$。y_0与此类似，不再赘述。通过设定x_0和y_0的值可以设置移动后B的左下角的位置。
- w_0和h_0的值分别为 $w_0=\frac{w_2}{w_1}$， $h_0=\frac{h_2}{h_1}$，通过设定w_0和h_0的值可以设置A物体被放缩的比例。当w_0与h_0相等时才能实现对A物体的等比例放缩。
- x_0和y_0的有效范围为[−1,1]；w_0和h_0的有效范围为[0,1]。

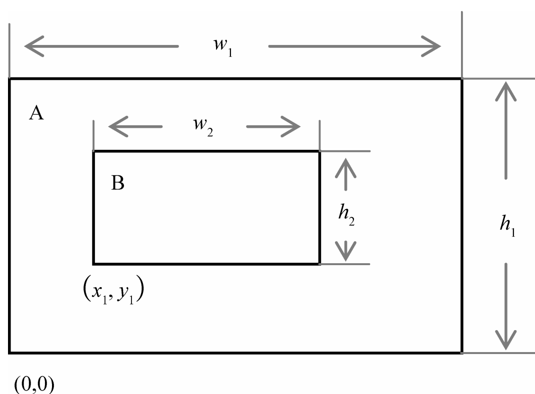


图2-6　rect放缩示意图

实例演示　下面通过实例演示属性rect的使用。

```
using UnityEngine;
using System.Collections;

public class Rect_ts : MonoBehaviour
{
    int which_change = -1;
    float temp_x = 0.0f, temp_y = 0.0f;
    void Update()
    {
        //视口平移
        if (which_change == 0)
        {
            if (camera.rect.x < 1.0f)
            {
                //沿着X轴平移
                temp_x = camera.rect.x + Time.deltaTime * 0.2f;
            }
            //取消下面注释查看平移的变化
            //if (camera.rect.y< 1.0f)
            //{
                //沿着y轴平移
            //    temp_y = camera.rect.y + Time.deltaTime * 0.2f
            //}
            camera.rect = new Rect(temp_x, temp_y, camera.rect.width, camera.rect.height);
        }
         //视口放缩
        else if (which_change == 1)
        {
            if (camera.rect.width > 0.0f)
            {
                //沿着X轴放缩
                temp_x = camera.rect.width - Time.deltaTime * 0.2f;
            }
            if (camera.rect.height > 0.0f)
            {
```

```
                //沿着y轴放缩
                temp_y = camera.rect.height - Time.deltaTime * 0.2f;
            }
            camera.rect = new Rect(camera.rect.x, camera.rect.y, temp_x, temp_y);
        }
    }
    void OnGUI()
    {
        if (GUI.Button(new Rect(10.0f, 10.0f, 200.0f, 45.0f), "视口平移"))
        {
            //重置视口
            camera.rect = new Rect(0.0f, 0.0f, 1.0f, 1.0f);
            which_change = 0;
            temp_y = 0.0f;
        }
        if (GUI.Button(new Rect(10.0f, 60.0f, 200.0f, 45.0f), "视口放缩"))
        {
            camera.rect = new Rect(0.0f, 0.0f, 1.0f, 1.0f);
            which_change = 1;
        }
        if (GUI.Button(new Rect(10.0f, 110.0f, 200.0f, 45.0f), "视口还原"))
        {
            camera.rect = new Rect(0.0f, 0.0f, 1.0f, 1.0f);
            which_change = -1;
        }
    }
}
```

在这段代码中，首先定义了一个变量which_change，然后在OnGUI方法中定义了3个按钮来更改which_change的值,最后在Update方法中根据不同的which_change值通过rect属性来实现不同的视口变换，包括平移和放缩。具体运行情况请自行运行程序查看。

2.1.11 renderingPath属性：渲染路径

基本语法 public RenderingPath renderingPath { get; set; }

功能说明 此属性用于获取或设置摄像机的渲染路径。Unity中渲染路径RenderingPath为枚举类型，共有以下4种设置方式。

- UsePlayerSettings：使用工程中的设置，即Edit→ProjectSettings→Player中各个平台下的设置，如图2-7所示。
- VertexLit：使用顶点光照，最低消耗的渲染路径，不支持实时阴影，适用于移动及老式设备。
- Forward：使用正向光照，基于着色器的渲染路径，支持逐像素计算光照（包括法线贴图和灯光Cookies）和来自一个平行光的实时阴影，具体请参考官方文档。
- DeferredLighting：使用延迟光照，支持实时阴影，计算消耗大，对硬件要求高，不支持移动设备，仅专业版可用。

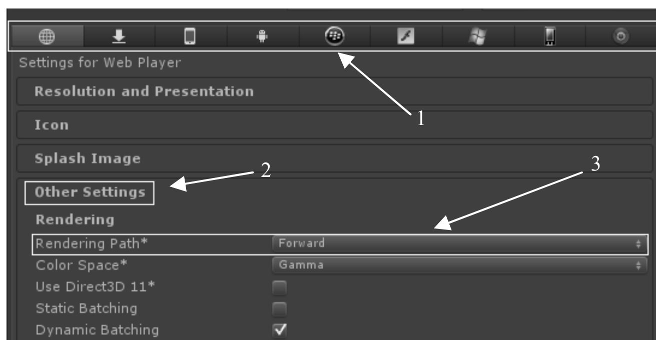


图2-7　使用UsePlayerSettings设置位置

实例演示　下面通过实例演示属性renderingPath的使用。

```
using UnityEngine;
using System.Collections;

public class renderingPath_ts : MonoBehaviour
{
    void OnGUI()
    {
        if (GUI.Button(new Rect(10.0f, 10.0f, 120.0f, 45.0f), "UsePlayerSettings"))
        {
            camera.renderingPath = RenderingPath.UsePlayerSettings;
        }
        if (GUI.Button(new Rect(10.0f, 60.0f, 120.0f, 45.0f), "VertexLit"))
        {
            //无凹凸纹理，无投影
            camera.renderingPath = RenderingPath.VertexLit;
        }
        if (GUI.Button(new Rect(10.0f, 110.0f, 120.0f, 45.0f), "Forward"))
        {
            //有凹凸纹理，只能显示一个投影
            camera.renderingPath = RenderingPath.Forward;
        }
        if (GUI.Button(new Rect(10.0f, 160.0f, 120.0f, 45.0f), "DeferredLighting"))
        {
            //有凹凸纹理，可以显示多个投影
            camera.renderingPath = RenderingPath.DeferredLighting;
        }
    }
}
```

这段代码的OnGUI方法中定义了4个Button，分别用于设置摄像机的不同renderingPath方式，请自行运行程序，查看在不同渲染模式下模型的凹凸纹理和投影的变化。

2.1.12　targetTexture属性：目标渲染纹理

基本语法　public RenderTexture targetTexture { get; set; }

功能说明 调用Camera此属性可以生成目标渲染纹理，仅专业版可用。此属性的作用是可以把某个摄像机A的视图作为Renderer Texture，然后添加到一个Material对象形成一个新的material，再把这个material赋给一个GameObject对象的Renderer组件（过程如图2-8和图2-9所示），便可以在这个GameObject中实时地看到摄像机A中的视图，此属性可以用来做一些实时跟踪类的功能。

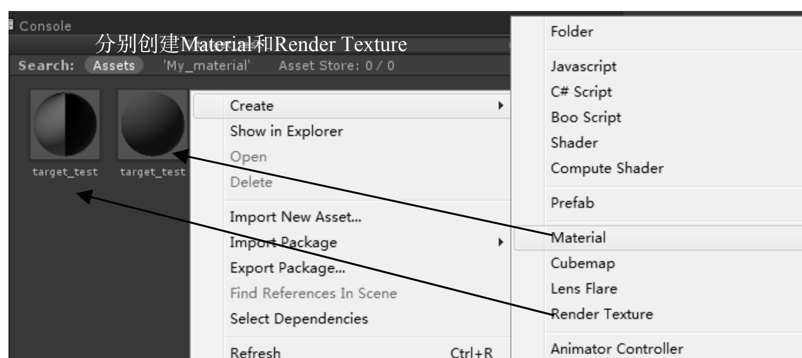


图2-8 targetTexture设置示意图01

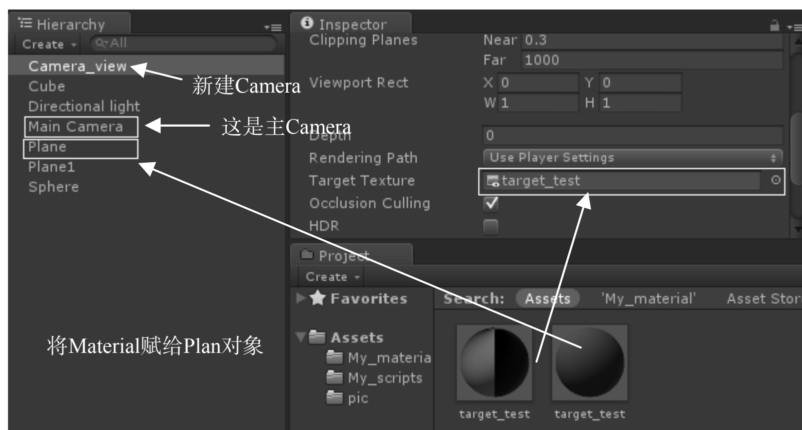


图2-9 targetTexture设置示意图02

实例演示 下面通过实例演示属性targetTexture的使用。

```
using UnityEngine;
using System.Collections;

public class Target_ts : MonoBehaviour {
    public Transform ts;
    void Update () {
        transform.RotateAround(ts.position,ts.up,30.0f*Time.deltaTime);
    }
}
```

这段代码的作用是让工程中摄像机Camera_view绕着物体Sphere旋转（需要在工程中将Sphere赋给ts变量）。targetTexture的设置过程如图2-8和图2-9所示，运行程序可以发现，在平面Plane上可以实时地显示摄像机Camera_view的视图。

2.1.13　worldToCameraMatrix属性：变换矩阵

基本语法　public Matrix4x4 worldToCameraMatrix { get; set; }

功能说明　此属性的功能是返回或设置从世界坐标系到当前Camera自身坐标系的变换矩阵。当用camera.worldToCameraMatrix重设摄像机的转换矩阵时，摄像机对应的Transform组件数据不会同步更新，如果想回到Transform的可控状态，需要调用ResetWorldToCameraMatrix方法重置摄像机的转换矩阵。

实例演示　下面通过实例演示属性worldToCameraMatrix的使用。

```
using UnityEngine;
using System.Collections;

public class WorldToCameraMatrix_ts : MonoBehaviour
{
    public Camera c_test;
    void OnGUI()
    {
        if (GUI.Button(new Rect(10.0f, 10.0f, 200.0f, 45.0f), "更改变换矩阵"))
        {
            //使用c_test的变换矩阵
            camera.worldToCameraMatrix = c_test.worldToCameraMatrix;
            //也可使用如下代码实现同样功能
            // camera.CopyFrom(c_test)
        }
        if (GUI.Button(new Rect(10.0f, 60.0f, 200.0f, 45.0f), "重置变换矩阵"))
        {
            //重置摄像机的转换矩阵，常和属性worldToCameraMatrix配合使用
            camera.ResetWorldToCameraMatrix();
        }
    }
}
```

在这段代码中，首先声明了一个Camera变量c_test，然后在OnGUI方法中定义了两个Button，点击Button“更改变换矩阵”，则会将c_test的变换矩阵赋给当前Camera，并且当前Camera的Transform组件也将失效；点击“重置变换矩阵”的Button，则当前Camera会使用原来的变换矩阵，Transform组件也会恢复有效性。如代码所示，也可以使用方法CopyFrom来实现同样的效果，但是在使用CopyFrom方法时，Transform组件不会失效。

2.2　Camera 类实例方法

在Camera类中，涉及的实例方法有RenderToCubemap、RenderWithShader、ScreenPointToRay、

ScreenToViewportPoint、ScreenToWorldPoint、SetTargetBuffers、ViewportPointToRay、ViewportToWorldPoint、WorldToScreenPoint和WorldToViewportPoint，下面详细介绍这些方法。

2.2.1 RenderToCubemap方法：生成Cubemap静态贴图

基本语法 (1) public bool RenderToCubemap(Cubemap cubemap);

其中参数cubemap为Cubemap静态贴图。

(2) public bool RenderToCubemap(RenderTexture cubemap);

其中参数cubemap为RenderTexture静态贴图。

(3) public bool RenderToCubemap(Cubemap cubemap, int faceMask);

其中参数cubemap为Cubemap静态贴图，faceMask 为反射面数量，默认值为63。

(4) public bool RenderToCubemap(RenderTexture cubemap, int faceMask);

其中参数cubemap为RenderTexture静态贴图，faceMask 为反射面数量，默认值为63。

功能说明 此方法的作用是使用摄像机生成一个Cubemap静态贴图。当faceMask值为63时，表示Cubemap的上下左右前后6个面全部反射，这种情况下系统计算耗费也最大。该值是一个二进制计算的参数，默认值为63即111111，表示6个面全开，如果不需要全部反射，则需要修改faceMask的值。

实例演示 下面通过实例演示方法RenderToCubemap的使用。

```
using UnityEngine;
using System.Collections;

public class RenderToCubemap_ts : MonoBehaviour
{
    public Cubemap cp;
    void Start()
    {
        camera.RenderToCubemap(cp);
    }
}
```

在实例工程中，需要先在Project面板中创建一个Cubemap对象（右键→Create→Cubemap），如本实例中的test02.cubemap（在pic文件夹下），再创建一个Material对象，如本实例中的six.mat（在My_material文件夹下），然后在Hierarchy面板中创建一个新的Camera，如本实例中的Camera_cubemap，并将脚本赋给这个Camera，用于生成Cubemap。最后只需要将Material赋给需要使用的物体即可，如本实例中的cubeMap物体。请自行在程序中查看，可以在Scene面板中查看cubeMap上各个面的反射状况，图2-10是本实例中物体cubeMap其中一个面的反射截图。

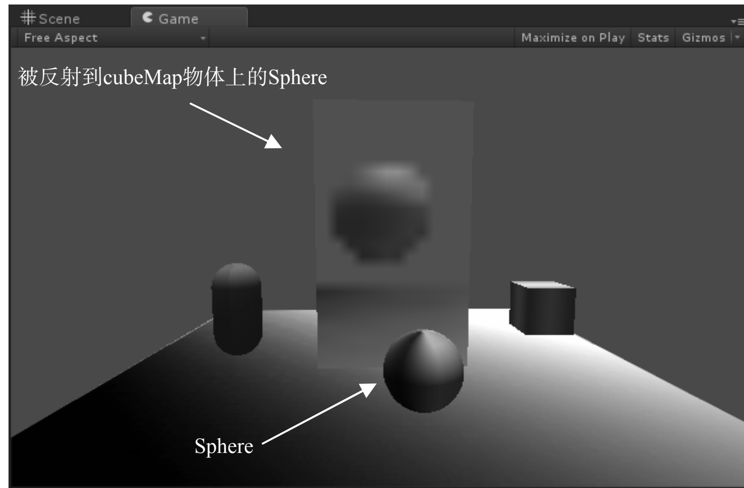


图2-10　cubeMap其中一个面的反射截图

2.2.2　RenderWithShader方法：使用其他shader渲染

基本语法　public void RenderWithShader(Shader shader, string replacementTag);

其中参数shader为要使用的shader，replacementTag 为shader的Tag标示。

功能说明　此方法的作用是可以使用指定的shader来代替当前物体的shader渲染一帧。当replacementTag为空时会替换视口中所有物体的shader。

提　示　SetReplacementShader方法与此方法功能相近，不同之处是，SetReplacementShader方法使用指定的shader来替换物体当前的shader，被替换后每一帧都会用替换的shader来渲染物体，而不是只渲染一帧，具体请查看实例演示。

实例演示　下面通过实例演示方法RenderWithShader的使用。

```
using UnityEngine;
using System.Collections;

public class RenderWithShader_ts : MonoBehaviour
{
    bool is_use = false;
    void OnGUI()
    {
        if (is_use)
        {
            //使用高光shader：Specular来渲染Camera
            camera.RenderWithShader(Shader.Find("Specular"), "RenderType");
        }
        if (GUI.Button(new Rect(10.0f, 10.0f, 300.0f, 45.0f), "使用RenderWithShader启用高光"))
```

```
            {
                //RenderWithShader每调用一次只渲染一帧，所以不可将其直接放到这儿
                //camera.RenderWithShader(Shader.Find("Specular"), "RenderType")
                is_use = true;
            }
            if (GUI.Button(new Rect(10.0f, 60.0f, 300.0f, 45.0f), "使用SetReplacementShader
                启用高光"))
            {
                //SetReplacementShader方法用来替换已有shader，调用一次即可
                camera.SetReplacementShader(Shader.Find("Specular"), "RenderType");
                is_use = false;
            }
            if (GUI.Button(new Rect(10.0f, 110.0f, 300.0f, 45.0f), "关闭高光"))
            {
                //重置摄像机的shader渲染模式
                camera.ResetReplacementShader();
                is_use = false;
            }
        }
    }
```

在这段代码中，首先定义了一个变量is_use用来记录是否启用高光shader，如果is_use为true，则在OnGUI方法中调用方法RenderWithShader来替换已有shader，也可直接使用方法SetReplacementShader来替换已有shader。如果要关闭高光，则只需重置shader即调用方法ResetReplacementShader()，并修改is_use值为false，具体显示情况请自行运行程序查看。

提　示 方法RenderWithShader每调用一次只渲染一帧，故不可直接将其放到GUI的Button中，否则看不出效果。

2.2.3 ScreenPointToRay方法：近视口到屏幕的射线

基本语法 public Ray ScreenPointToRay(Vector3 position);

其中参数position为屏幕位置参考点。

功能说明 此方法的作用是可以从Camera的近视口nearClip向前发射一条射线到屏幕上的position点。参考点position用实际像素值的方式来决定Ray到屏幕的位置。参考点position的x轴分量或y轴分量从0增长到最大值时，Ray从屏幕一边移动到另一边。当Ray未能碰撞到物体时，hit.point返回值为Vector3(0,0,0)。参考点position的z轴分量值无效。

实例演示 下面通过实例演示方法ScreenPointToRay的使用。

```
using UnityEngine;
using System.Collections;

public class ScreenPointToRay_ts : MonoBehaviour
```

```
{
    Ray ray;
    RaycastHit hit;
    Vector3 v3 = new Vector3(Screen.width / 2.0f, Screen.height / 2.0f, 0.0f);
    Vector3 hitpoint = Vector3.zero;
    void Update()
    {
        //射线沿着屏幕X轴从左向右循环扫描
        v3.x = v3.x >= Screen.width ? 0.0f : v3.x + 1.0f;
        //生成射线
        ray = camera.ScreenPointToRay(v3);
        if (Physics.Raycast(ray, out hit, 100.0f))
        {
            //绘制线，在Scene视图中可见
            Debug.DrawLine(ray.origin, hit.point, Color.green);
            //输出射线探测到的物体的名称
            Debug.Log("射线探测到的物体名称：" + hit.transform.name);
        }
    }
}
```

在这段代码中，首先声明了一个变量v3，用于记录射线到屏幕上的实际像素坐标，然后在Update方法中更改v3的x分量值，使得射线从屏幕左方向右方不断循环扫描，接着调用方法ScreenPointToRay生成射线ray，最后绘制射线和打印射线探测到的物体的名称。如果想要观察绘制的射线，请运行程序后在Scene视图中查看，图2-11是射线探测到的屏幕上物体名称的输出。

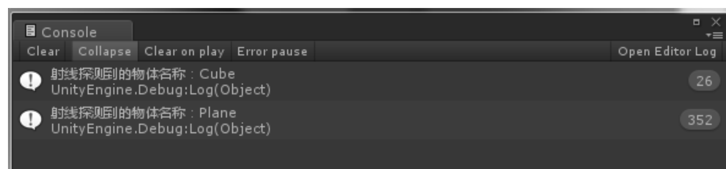


图2-11　射线探测到的物体的名称

2.2.4　ScreenToViewportPoint方法：坐标系转换

基本语法　`public Vector3 ScreenToViewportPoint(Vector3 position);`

其中参数position为屏幕参考点。

功能说明　此方法的功能是实现坐标点position从屏幕坐标系向摄像机视口的单位化坐标系转换。参考点position的x和y分量为屏幕的实际坐标值，单位为像素，z值无效。

实例演示　下面通过实例演示方法ScreenToViewportPoint的使用。

```
using UnityEngine;
using System.Collections;

public class ScreenToViewportPoint_ts : MonoBehaviour
```

```
{
    void Start()
    {
        transform.position = new Vector3(0.0f, 0.0f, 1.0f);
        transform.rotation = Quaternion.identity;
        //从屏幕的实际坐标点向视口的单位化比例值转换
        Debug.Log("1:" + camera.ScreenToViewportPoint(new Vector3(Screen.width / 2.0f,
            Screen.height / 2.0f, 100.0f)));
        //从视口的单位化比例值向屏幕的实际坐标点转换
        Debug.Log("2:" + camera.ViewportToScreenPoint(new Vector3(0.5f, 0.5f, 100.0f)));
        Debug.Log("屏幕宽：" + Screen.width + "  屏幕高：" + Screen.height);
    }
}
```

在这段代码中，首先重置了摄像机的position和rotation，然后调用方法ScreenToViewportPoint将屏幕的正中间位置转换为视口比例值并打印出来，接着调用方法ViewportToScreenPoint将视口的中间位置转换为屏幕的实际像素值并打印出来，最后打印出了屏幕的宽度和高度，程序运行结果如图2-12所示。

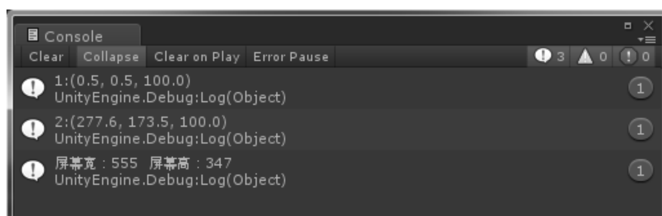


图2-12 ScreenToViewportPoint实例演示的运行结果

2.2.5 ScreenToWorldPoint方法：坐标系转换

基本语法 public Vector3 ScreenToWorldPoint(Vector3 position);

其中参数position为屏幕参考点。

功能说明 此方法的作用是将参考点position从屏幕坐标系转换到世界坐标系。此方法与方法ViewportToWorldPoint功能类似，只是此方法的参考点position中各个分量值都为实际单位像素值，而非比例值。

例如执行如下代码后，

```
Vector v3= camera. ScreenToWorldPoint (ps);//ps为已知参考点
```

v3的各个分量值为：

v3.x= camera.transform.position.x+ ps.z*asp* tan(e/2);
v3.y= camera.transform.position.y+ ps.z* tan(e/2);
v3.z= camera.transform.position.z +ps.z;

其中ps为已知参考值，e为摄像机的视口夹角fieldOfView的值，asp为摄像机视口的宽

高比例值aspect。更多内容请参考方法ViewportToWorldPoint的功能说明。

实例演示　下面通过实例演示方法ScreenToWorldPoint的使用。

```
using UnityEngine;
using System.Collections;

public class ScreenToWorldPoint_ts : MonoBehaviour
{
    void Start()
    {
        transform.position = new Vector3(0.0f, 0.0f, 1.0f);
        camera.fieldOfView = 60.0f;
        camera.aspect = 16.0f / 10.0f;
        //z轴前方100处对应的屏幕左下角的世界坐标值
        Debug.Log("1:" + camera.ScreenToWorldPoint(new Vector3(0.0f, 0.0f, 100.0f)));
        //z轴前方100处对应的屏幕中间的世界坐标值
        Debug.Log("2:" + camera.ScreenToWorldPoint(new Vector3(Screen.width / 2.0f,
            Screen.height / 2.0f, 100.0f)));
        //z轴前方100处对应的屏幕右上角的世界坐标值
        Debug.Log("3:" + camera.ScreenToWorldPoint(new Vector3(Screen.width, Screen.height,
            100.0f)));
    }
}
```

在这段代码中，首先重置了摄像机的位置position、视口夹角fieldOfView和视口宽高比aspect，然后调用方法ScreenToWorldPoint分别计算和打印出了视口前方100米处的左下角、中间和右上角的坐标值，程序运行结果如图2-13所示。

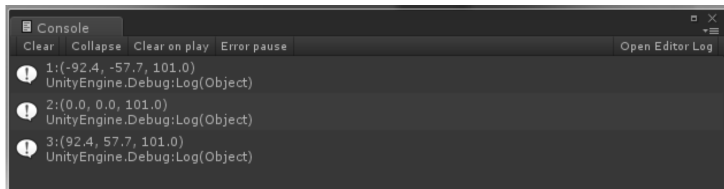


图2-13　ScreenToWorldPoint实例演示的运行结果

2.2.6　SetTargetBuffers方法：重设摄像机到TargetTexture的渲染

基本语法　(1) public void SetTargetBuffers(RenderBuffer colorBuffer, RenderBuffer depth Buffer);

其中参数colorBuffer为纹理的颜色缓存，depthBuffer为纹理的深度缓存。

(2) public void SetTargetBuffers(RenderBuffer[] colorBuffer, RenderBuffer depth Buffer);

其中参数colorBuffer为纹理的颜色缓存，depthBuffer为纹理的深度缓存。此重载方法可以将摄像机的渲染一次赋给多个colorBuffer。

功能说明　此方法用于将Camera的渲染赋给RenderTexture的colorBuffer和depthBuffer。

实例演示 下面通过实例演示方法SetTargetBuffers的使用。在本实例演示中，先在Project面板的My_material文件夹下创建了一个setbuffer.mat和两个RenderTexture，并在场景中创建两个辅助摄像机和一些物体，其设置类似于属性TargetTexture中的设置方法。然后将如下脚本赋给MainCamera，如图2-14所示。

```
using UnityEngine;
using System.Collections;

public class SetTargetBuffers_ts : MonoBehaviour
{
    //声明两个RendererTexture变量
    public RenderTexture RT_1, RT_2;
    public Camera c;//指定Camera

    void OnGUI()
    {
        //设置RT_1的buffer为摄像机c的渲染
        if (GUI.Button(new Rect(10.0f, 10.0f, 180.0f, 45.0f), "set target buffers"))
        {
            c.SetTargetBuffers(RT_1.colorBuffer, RT_1.depthBuffer);
        }
        //设置RT_2的buffer为摄像机c的渲染，此时RT_1的buffer变为场景中Camera1的渲染
        if (GUI.Button(new Rect(10.0f, 60.0f, 180.0f, 45.0f), "Reset target buffers"))
        {
            c.SetTargetBuffers(RT_2.colorBuffer, RT_2.depthBuffer);
        }
    }
}
```

在这段代码中，首先声明了两个RenderTexture类型的变量和一个Camera变量，然后在OnGUI方法中定义了两个Button，用于演示Camera渲染到不同RendererTexture的变化，请自行运行程序，查看不同状态下场景中物体show上的变化。

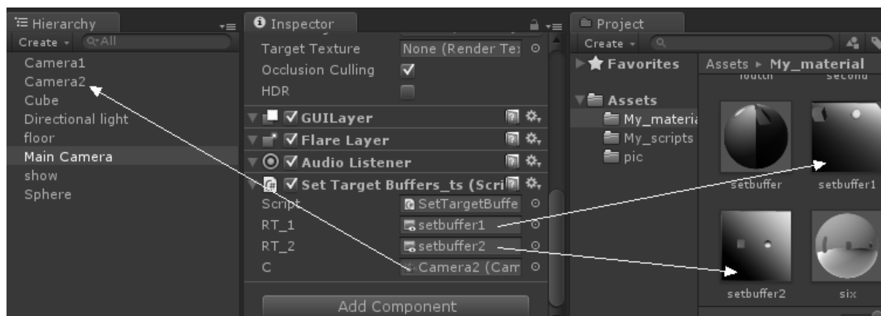


图2-14 脚本中变量的设置

2.2.7 ViewportPointToRay方法：近视口到屏幕的射线

基本语法 `public Ray ViewportPointToRay(Vector3 position);`

其中参数position为单位化坐标中的参考点。

功能说明 此方法的作用是可以从Camera的近视口nearClip向前发射一条射线到屏幕上的position点。参考点position用单位化比例值的方式来决定Ray到屏幕的位置。参考点position的x轴分量或y轴分量从0到1增长时，Ray从屏幕一边移动到另一边。当Ray未能碰撞到物体时，hit.point的返回值为Vector3(0,0,0)。参考点position的z轴分量值无效。

实例演示 下面通过实例演示方法ViewportPointToRay的使用。

```
using UnityEngine;
using System.Collections;

public class ViewportPointToRay_ts : MonoBehaviour
{
    Ray ray;//射线
    RaycastHit hit;
    Vector3 v3 = new Vector3(0.5f, 0.5f, 0.0f);
    Vector3 hitpoint = Vector3.zero;
    void Update()
    {
        //射线沿着屏幕x轴从左向右循环扫描
        v3.x = v3.x >= 1.0f ? 0.0f : v3.x + 0.002f;
        //生成射线
        ray = camera.ViewportPointToRay(v3);
        if (Physics.Raycast(ray, out hit, 100.0f))
        {
            //绘制线，在Scene视图中可见
            Debug.DrawLine(ray.origin, hit.point, Color.green);
            //输出射线探测到的物体的名称
            Debug.Log("射线探测到的物体名称：" + hit.transform.name);
        }
    }
}
```

在这段代码中，首先声明了一个Vector3类型的变量v3，用于记录射线到屏幕上的视口比例位置，然后在Update方法中更改v3的x分量值，使得射线从屏幕左方向右方不断循环扫描，接着调用方法ViewportPointToRay生成射线ray，最后绘制射线和打印射线探测到的物体的名称。如果想要观察绘制的射线，请运行程序后在Scene视图中查看，图2-15是射线探测到的物体名称的输出结果。

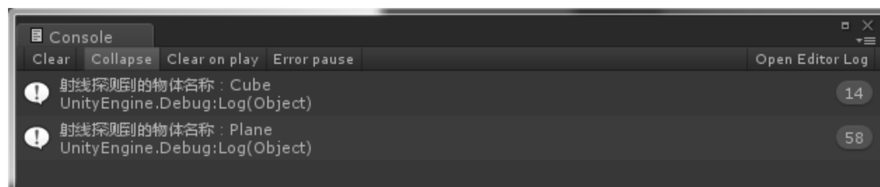


图2-15 射线探测到的物体的名称

2.2.8 ViewportToWorldPoint方法：坐标点的坐标系转换

基本语法 public Vector3 ViewportToWorldPoint(Vector3 position);

其中参数position为待转换的参考点。

功能说明 此方法的功能是实现从Camera视口坐标点向世界坐标点转换，这与方法WorldToViewportPoint的功能正好相反。此方法的返回值大小受当前Camera在世界坐标系中的位置Camera的fieldOfView值以及参考点position的共同影响。其中参考点position的x和y分量的有效范围为[0.0,1.0]，为比例值；而z值为实际单位值，非比例值。

对此方法的算法说明如下。

设o为摄像机坐标点（如图2-16所示），asp为摄像机的视口宽高比例值aspect，e为摄像机的视口夹角fieldOfView值，oA的模长为参考点position的z值，参考点position的y轴分量值为y_0，则此方法执行后的返回值的z值大小为：o点的z轴分量值加上oA的长度；返回值的y分量值为o点的y轴分量值加上K_1，其中K_1的计算方法为：

$$K_1=|oA|*((y_0-0.5)/0.5)*\tan(e/2);$$

返回值的x分量值为o点的x轴分量值加上K_2，其中K_2的计算方法为：

$$K_2=|oA|*asp*((y_0-0.5)/0.5)*\tan(e/2);$$

例如执行如下代码后，

```
Vector v3= camera.ViewportToWorldPoint(ps);//ps为已知参考点
```

v3的各个分量值为：

v3.x= camera.transform.position.x+ ps.z*asp*((ps.x–0.5)/0.5)* tan(e/2);
v3.y= camera.transform.position.y+ ps.z*((ps.y–0.5)/0.5)* tan(e/2);
v3.z= camera.transform.position.z +ps.z;

其中ps为已知参考值，e为摄像机的视口夹角fieldOfView的值，asp为摄像机视口的宽高比例值aspect。

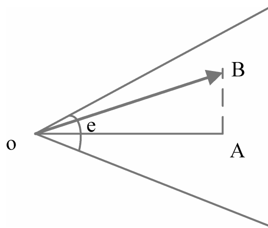


图2-16 Camera的视口示意图

实例演示 下面通过实例演示方法ViewportToWorldPoint的使用。

```
using UnityEngine;
using System.Collections;

public class ViewportToWorldPoint_ts : MonoBehaviour
{
    void Start()
    {
        transform.position = new Vector3(1.0f, 0.0f, 1.0f);
        camera.fieldOfView = 60.0f;
        camera.aspect = 16.0f / 10.0f;
        //屏幕左下角
        Debug.Log("1:" + camera.ViewportToWorldPoint(new Vector3(0.0f, 0.0f, 100.0f)));
        //屏幕中间
        Debug.Log("2:" + camera.ViewportToWorldPoint(new Vector3(0.5f, 0.5f, 100.0f)));
        //屏幕右上角
        Debug.Log("3:" + camera.ViewportToWorldPoint(new Vector3(1.0f, 1.0f, 100.0f)));
    }
}
```

在这段代码中，首先重置了摄像机的位置position、视口夹角fieldOfView和视口宽高比aspect，然后调用方法ViewportToWorldPoint分别计算和打印出了视口前方100米处的左下角、中间和右上角的坐标值，程序运行结果如图2-17所示。对输出结果的计算方法请参考功能说明。

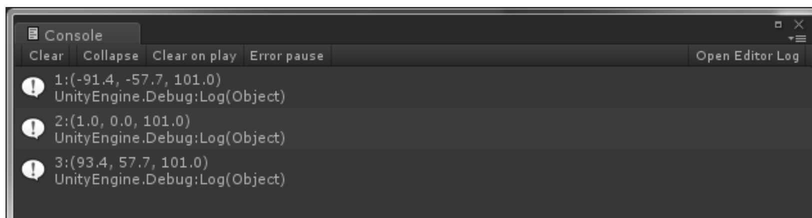


图2-17　ViewportToWorldPoint实例演示的运行结果

2.2.9　WorldToScreenPoint方法：坐标点的坐标系转换

基本语法　public Vector3 WorldToScreenPoint(Vector3 position);

其中参数position为待转换的世界坐标系中的坐标点。

功能说明　此方法用来实现从世界坐标点向屏幕坐标点转换，即坐标点position投射到屏幕上的坐标值。返回值的x和y分量是以屏幕左下角为(0,0)点，以向上为y轴、向右为x轴来计算的。

实例演示　下面通过实例演示方法WorldToScreenPoint的使用。

```
using UnityEngine;
using System.Collections;

public class WorldToScreenPoint_ts : MonoBehaviour
{
    public Transform cb, sp;
    public Texture2D t2;
    Vector3 v3 = Vector3.zero;
    float sg;
    void Start()
    {
        //记录屏幕高度
        sg = Screen.height;
    }
    void Update()
    {
        //sp绕着cb的y轴旋转
        sp.RotateAround(cb.position, cb.up, 30.0f * Time.deltaTime);
        //获取sp在屏幕上的坐标点
        v3 = camera.WorldToScreenPoint(sp.position);
    }
    void OnGUI()
    {
        //绘制纹理
        GUI.DrawTexture(new Rect(0.0f, sg - v3.y, v3.x, sg), t2);
    }
}
```

在这段代码中，首先在Update方法中让物体sp绕着cb的y轴旋转，再通过方法WorldToScreenPoint获取物体sp映射到屏幕上的位置v3，最后在OnGUI方法中，在由屏幕左下角(0,0)点和v3点组成的矩形区域内绘制纹理，图2-18是一张运行时截图。

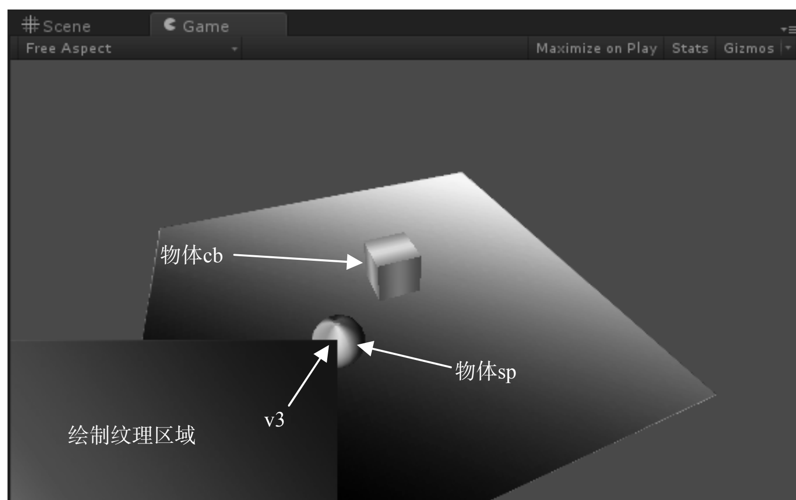


图2-18　方法WorldToScreenPoint的实例演示运行时截图

2.2.10　WorldToViewportPoint方法：坐标点的坐标系转换

基本语法　public Vector3 WorldToViewportPoint(Vector3 position);

其中参数position 为待转换的世界坐标系中的坐标点。

功能说明　此方法的功能是把三维坐标点position从世界坐标系转换到屏幕的单位化坐标系中，即世界坐标点position投射到屏幕上的坐标点的x、y分量所占屏幕宽高的比例大小。此方法与方法WorldToScreenPoint功能类似，不同的是其返回值的x和y分量是比例值，以屏幕的总宽度和总高度分别为x和y分量的最大值1。返回值的x和y分量是以屏幕左下角为(0,0)点，以向上为y轴、向右为x轴来计算的。

实例演示　下面通过实例演示方法WorldToViewportPoint的使用。

```
using UnityEngine;
using System.Collections;

public class WorldToViewportPoint_ts : MonoBehaviour
{
    public Transform cb, sp;
    public Texture2D t2;
    Vector3 v3 = Vector3.zero;
    float sw, sh;
    void Start()
    {
        //记录屏幕的宽度和高度
        sw = Screen.width;
        sh = Screen.height;
    }
    void Update()
    {
        //物体sp始终绕cb的y轴旋转
        sp.RotateAround(cb.position, cb.up, 30.0f * Time.deltaTime);
        //记录sp映射到屏幕上的比例值
        v3 = camera.WorldToViewportPoint(sp.position);
    }
    void OnGUI()
    {
        //绘制纹理，由于方法WorldToViewportPoint的返回值的y分量是从屏幕下方向上方递增的，
        //所以需要先计算1.0f - v3.y的值，然后再和sh相乘
        GUI.DrawTexture(new Rect(0.0f, sh * (1.0f - v3.y), sw * v3.x, sh), t2);
    }
}
```

在这段代码中，首先声明两个变量sw和sh用来记录屏幕的宽度和高度，然后在Update方法中让物体sp绕着cb的y轴旋转，再通过方法WorldToViewportPoint获取物体sp映射到屏幕上的单位化坐标值v3，最后在OnGUI方法中，在由屏幕左下角(0,0)点和v3点组成的矩形区域内绘制纹理。请自行运行程序查看程序运行状况。

2.3 关于Camera视口、aspect、pixelRect及rect的关系注解

搞清楚摄像机的视口与aspect、pixelRect及rect之间的关系有助于对Camera类的理解和使用，下面对它们之间的关系进行说明。

- Camera视口用来记录当前摄像机能看到场景中的哪些内容，其大小及位置是可以改变的。而屏幕视口是指当前硬件的屏幕，对于一个固定的硬件（如手机），它的屏幕视口大小（即分辨率）是固定的。Camera视口的内容不一定可以完全显示在屏幕上，屏幕可能只显示了一部分视口内容，也可能对视口内容进行了放缩。可以简单理解为，Camera视口是一张二维图片，而屏幕是用来显示这张图片的，图片可能被剪切，也可能被压缩。Camera视口的内容显示到屏幕上的方式由很多因素决定。
- Unity的Game面板中的aspect选项（如图2-19所示）是用来模拟硬件屏幕的，可分为3类：全屏显示、固定比例显示和固定分辨率显示。全屏方式即以当前Game屏幕的大小来模拟硬件屏幕分辨率，其Camera视口即为当前摄像机的默认状态。而在固定比例方式则会改变Camera视口的宽高比例，其大小不固定。而在固定分辨率方式下，视口的最大宽度和高度是固定的，当Game视口的宽度和高度大于固定分辨率时，其有效显示区间将保持固定分辨率的大小（如图2-20所示）。

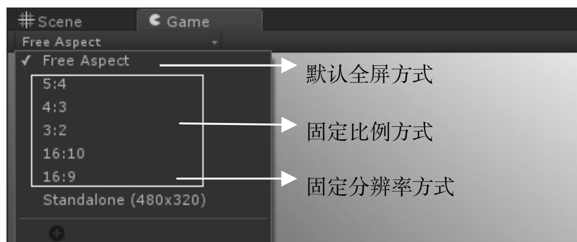


图2-19 Camera屏幕分辨率的设置方式

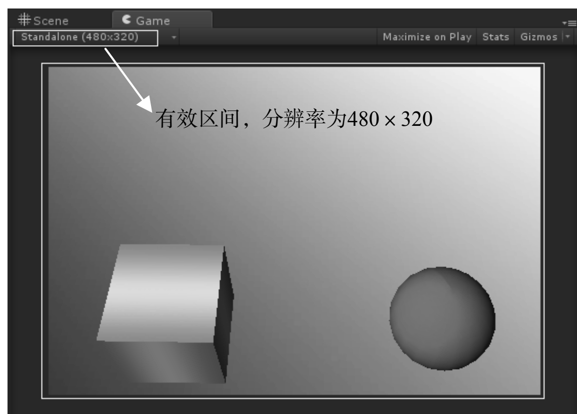


图2-20 Camera固定分辨率方式下的有效区间

- 在Camera.aspect固定的情况下，无论选择Game视图中哪种屏幕模拟方式，它们的显示内容都是相同的（请查看实例演示）。不同的屏幕模拟方式只会对显示的内容进行放缩。决定屏幕视口显示内容的是Camera.aspect的值和Camera的Transform，至于屏幕要如何显示Camera视口的内容，那就是硬件显示屏要处理的事情了。
- PixelRect和Rect功能类似，都是决定硬件显示屏如何显示Camera视口提供的内容的。不同的是，PixelRect以实际像素来展示显示内容，而Rect以单位化形式来展示显示内容（请查看PixelRect和Rect的相关实例演示）。

实例演示　下面通过实例演示Camera视口和aspect的关系。

```
using UnityEngine;
using System.Collections;

public class Compare_ts : MonoBehaviour
{
    Vector3 v1;
    void Start()
    {
        v1 = transform.position;
    }
    void OnGUI()
    {
        //设置Camera视口宽高比例为2：1，点击此按钮后更改Game视图中不同的aspect值
        //尽管Scene视图中Camera视口的宽高比会跟着改变
        //但实际Camera视口的内容依然是按照2：1的比例获取的
        //不同的屏幕显示相同的内容会发生变形
        if (GUI.Button(new Rect(10.0f, 10.0f, 200.0f, 45.0f), "Camera视口宽高比例2：1"))
        {
            camera.aspect = 2.0f;
            transform.position = v1;
        }
        if (GUI.Button(new Rect(10.0f, 60.0f, 200.0f, 45.0f), "Camera视口宽高比例1：2"))
        {
            camera.aspect = 0.5f;
            //更改Camera坐标，使被拉伸后的物体显示出来
            transform.position = new Vector3(v1.x, v1.y, 333.2f);
        }
        //恢复aspect的默认设置，即屏幕比例和Camera视口比例保持一致
        if (GUI.Button(new Rect(10.0f, 110.0f, 200.0f, 45.0f), "使用Game面板中aspect的选择"))
        {
            camera.ResetAspect();
            transform.position = v1;
        }
    }
}
```

在这段代码中，用3个Button来设置3种不同的Camera视口比例，在实例工程中设置Game视图的aspect值为16：10。如图2-21所示，按钮“使用Game面板中aspect的选择”的作用即为使视口的宽高比为Game视图中的设置，在这种状态下，场景中物体不会变形。

当设置Camera视口宽高比例为2：1时，相当于将Camera的视口变宽（2：1 > 16：10），Camera的x轴方向上的视野将相对更大，要将更大的视野放到相同大小的屏幕上，物体自然会被压缩，如图2-22所示。同理，当设置Camera视口宽高比例为1：2时，相当于将Camera的视口变窄（1：2 < 16：10），要将更小的视野放到相同大小的屏幕上，物体自然会被拉伸，如图2-23所示。

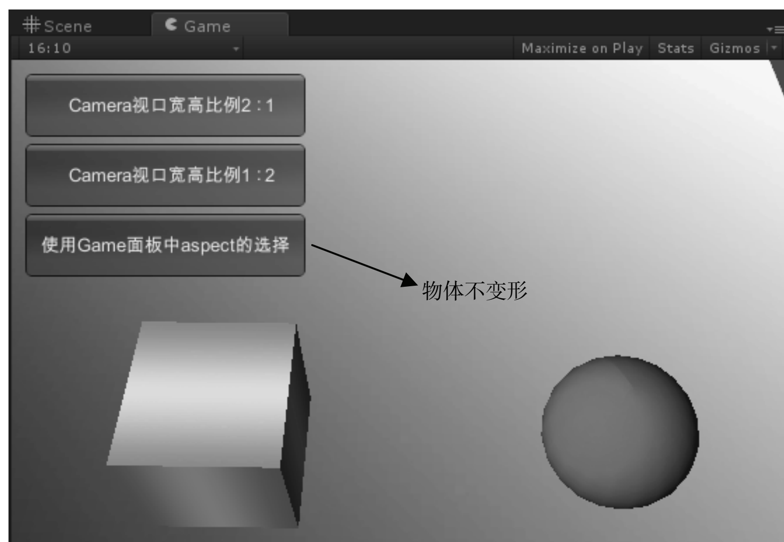


图2-21　使用Game面板中aspect的选择时

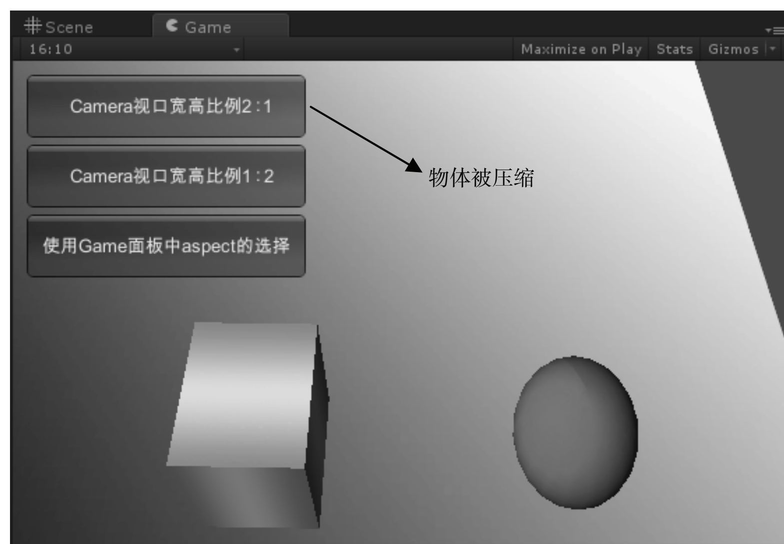


图2-22　Camera视口宽高比例为2：1时



图2-23　Camera视口宽高比例为1：2时

第 3 章

GameObject类

GameObject类是Unity场景中所有实体的基类。一个GameObject对象通常由多个组件（component）组成，且至少含有一个Transform组件。本章主要介绍GameObject类的一些实例属性、构造方法、实例方法和静态方法，并在最后对GameObject和Component这两个类之间的关系及其涉及的GetComponent相关方法的使用区别进行了注解。

3.1 GameObject 类实例属性

在GameObject类中，涉及的实例属性有activeSelf和activeInHierarchy。由于这两个属性功能相似，因此将属性activeInHierarchy作为activeSelf属性的提示内容介绍，下面主要介绍activeSelf属性。

activeSelf属性：GameObject的Active标识

基本语法 public bool activeSelf { get; }

功能说明 此属性用来返回GameObject对象的Active标识状态，如图3-1所示。

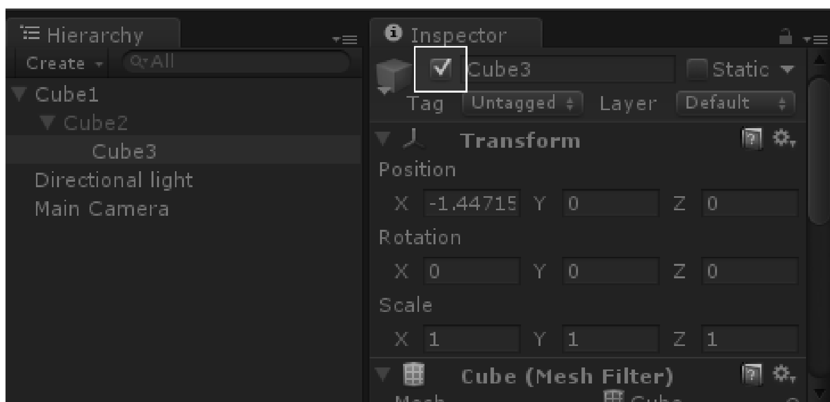


图3-1 GameObject对象自身Active标示

在图3-1中，cube1、cube2和cube3的Active状态分别为true、false和true。由于cube2的Active为false，作为其子类cube3虽然没有被激活，但其activeSelf的返回值依然为true。

提　　示　注意此属性与属性activeInHierarchy的区别。activeInHierarchy属性的功能是返回GameObject实例在程序运行时的激活状态，它只有当GameObject实例的状态被激活时才会返回true。而且它会受其父类对象激活状态的影响，如果其父类至最顶层的对象中有一个对象未被激活，activeInHierarchy就会返回false。

实例演示　下面通过实例演示属性activeSelf 和activeInHierarchy在功能上的不同。

```
using UnityEngine;
using System.Collections;

public class ActiveSelf_ts : MonoBehaviour {
    public GameObject cube1, cube2, cube3;
    void Start () {
        //对cube2设置为false，其他设置为true
        cube1.SetActive(true);
        cube2.SetActive(false);
        cube3.SetActive(true);

        Debug.Log("activeSelf:");
        //尽管cube2被设置为false，但其子类cube3的activeSelf返回值仍然为true
        Debug.Log("cube1.activeSelf:" + cube1.activeSelf);
        Debug.Log("cube2.activeSelf:" + cube2.activeSelf);
        Debug.Log("cube3.activeSelf:" + cube3.activeSelf);

        Debug.Log("\nactiveInHierarchy:");
        //cube2和cube3的activeInHierarchy返回值都为false
        Debug.Log("cube1.activeInHierarchy:" + cube1.activeInHierarchy);
        Debug.Log("cube2.activeInHierarchy:" + cube2.activeInHierarchy);
        Debug.Log("cube3.activeInHierarchy:" + cube3.activeInHierarchy);
    }
}
```

在这段代码中，首先声明了3个变量cube1、cube2和cube3分别指向实例工程中的3个GameObject对象cube1、cube2和cube3，这3个对象的层级关系如图3-1所示。然后分别设置cube1、cube2和cube3的Active为true、false和true。最后分别打印出这3个对象的activeSelf属性值和activeIn Hierarchy属性值，结果如图3-2所示。从输出结果可以发现属性activeSelf和activeInHierarchy功能的不同之处，详见代码注释。

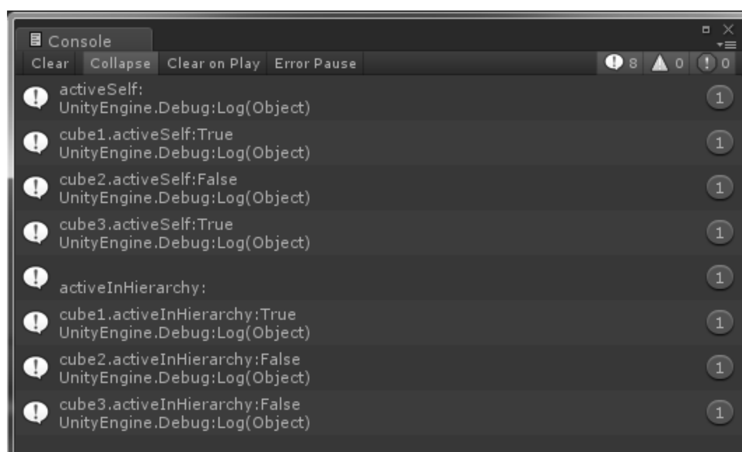


图3-2　activeSelf实例演示的运行结果

3.2　GameObject 构造方法

基本语法　(1) public GameObject();

(2) public GameObject(string name);

其中参数name为构造GameObject对象的名字。

(3) public GameObject(string name, params Type[] components);

其中参数name为构造GameObject对象的名字，components为构造对象要添加的组件类型集合，多个组件之间用逗号隔开。

功能说明　此构造方法用来创建一个GameObject对象。

实例演示　下面通过实例演示GameObject的3种不同构造函数的使用方法。

```
using UnityEngine;
using System.Collections;

public class Constructors_ts : MonoBehaviour {
    void Start () {
        //使用构造函数GameObject (name : String)
        GameObject g1 = new GameObject("G1");
        g1.AddComponent<Rigidbody>();
        //使用构造函数GameObject ()
        GameObject g2 = new GameObject();
        g2.AddComponent<FixedJoint>();
        //使用构造函数GameObject (name : String, params components : Type[])
        GameObject g3 = new GameObject("G3",typeof(MeshRenderer),typeof(Rigidbody),typeof
            (SpringJoint));

        Debug.Log("g1 name:" + g1.name + "\nPosition:" + g1.transform.position);
```

```
        Debug.Log("g2 name:" + g2.name + "\nPosition:" + g2.transform.position);
        Debug.Log("g3 name:" + g3.name + "\nPosition:" + g3.transform.position);
    }
}
```

在这段代码中，分别演示了GameObject的3种不同构造函数的使用方法，并打印出生成的GameObject对象的名称和位置，如图3-3所示。

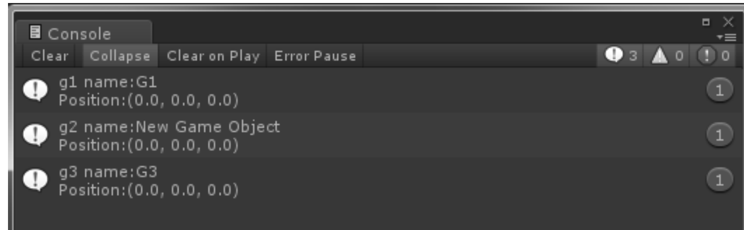


图3-3　GameObject的构造方法的实例演示的运行结果

3.3　GameObject 类实例方法

在GameObject类中，涉及的实例方法有GetComponent、GetComponentInChildren、GetComponents、GetComponentsInChildren、SendMessage、BroadcastMessage和SendMessageUpwards。由于GetComponent、GetComponentInChildren、GetComponents和GetComponentsInChildren这4个方法都和获取GameObject对象的组件有关，因此将它们放在一起介绍；而方法SendMessage、BroadcastMessage和SendMessageUpwards都和发送消息有关，于是也将它们放在一起介绍。下面一一介绍这些方法。

3.3.1　GetComponent方法：获取组件

基本语法　(1) public T GetComponent<T>() where T : Component;

(2) public Component GetComponent(string type);

其中参数type为组件名。

(3) public Component GetComponent(Type type);

其中参数type为组件类型。

功能说明　此方法用于获取GameObject中第一个符合Type类型的Component。

提　　示　与此方法功能相似的方法有GetComponentInChildren、GetComponents和GetComponentsInChildren，它们的具体使用请参考实例演示。需要注意以下两点。

- 在使用 GetComponents (type : Type)方法时，

```
Component[] cjs = GetComponents(typeof(ConfigurableJoint)) as Component[];
```

注意不可以这样写：

```
ConfigurableJoint[] cjs = GetComponents(typeof(ConfigurableJoint)) as Configurable Joint [];
```

因为ConfigurableJoint不是Component，而是其子类，建议使用其泛型方式。

- 在使用 GetComponentsInChildren (type : Type, includeInactive : boolean = false) 方法时，

  ```
  Component[] cjs = GetComponentsInChildren(typeof(ConfigurableJoint), false) as Component[];
  ```

  注意不可以这样写：

  ```
  ConfigurableJoint[] cjs =
  GetComponentsInChildren(typeof(ConfigurableJoint), false) as ConfigurableJoint[];
  ```

  因为ConfigurableJoint不是Component，而是其子类，建议使用其泛型方式。

3

实例演示 下面通过实例演示方法GetComponent、GetComponentInChildren、GetComponents和GetComponentsInChildren的使用。

```
using UnityEngine;
using System.Collections;

public class GetComponent_ts : MonoBehaviour {
    void Start () {

/**
 *以下是GetComponent方法的相关使用代码
 **/
        Debug.Log("以下是GetComponent的相关使用代码。\nGetComponent方法用来获取当前
            GameObject中符合Type类型的第一个组件。");
        //GetComponent (type : Type)
        Rigidbody rb = GetComponent(typeof(Rigidbody))as Rigidbody;
        Debug.Log("使用GetComponent (type : Type)获取Rigidbody：" + rb.GetInstanceID());

        //GetComponent.<T>()
        rb=GetComponent<Rigidbody>();
        Debug.Log("使用GetComponent.<T>()获取Rigidbody：" + rb.GetInstanceID());

        //GetComponent (type : String)
        rb = GetComponent("Rigidbody") as Rigidbody;
        Debug.Log("使用GetComponent (type : String)获取Rigidbody：" + rb.GetInstanceID());

/**
 *以下是GetComponentInChildren方法的相关使用代码
 **/
        Debug.Log("以下是GetComponentInChildren的相关使用代码。\nGetComponentInChildren方法
            用来获取当前GameObject的所有子类中中符合Type类型的第一个组件。");

        //GetComponentInChildren (type : Type)
```

```
        //从GameObject自身及其子类中返回第一个符合type类型的Component
        rb = GetComponentInChildren(typeof(Rigidbody)) as Rigidbody;
        Debug.Log("使用GetComponentInChildren (type : Type)获取Rigidbody：" + rb.name);

        //GetComponentInChildren.<T> ()
        rb=GetComponentInChildren<Rigidbody>();
        Debug.Log("使用GetComponentInChildren.<T>()获取Rigidbody：" + rb.name);

  /**
   *以下是GetComponents方法的相关使用代码
   **/
        Debug.Log("以下是GetComponents的相关使用代码。\nGetComponents方法用来获取当前
            GameObject中符合Type类型的所有组件。");

        //GetComponents (type : Type)
        //返回GameObject中所有符合type类型的Component集合
        Component[] cjs = GetComponents(typeof(ConfigurableJoint)) as Component[];
        foreach(ConfigurableJoint cj in cjs){
            Debug.Log("使用GetComponents (type : Type)获取ConfigurableJoint：" + cj.GetInstance
                ID());
        }

        //GetComponents.<T> ()
        cjs = GetComponents<ConfigurableJoint>();
        foreach (ConfigurableJoint cj in cjs)
        {
            Debug.Log("使用GetComponents.<T>()获取ConfigurableJoint：" + cj.GetInstanceID());
        }

  /**
   *以下是GetComponentsInChildren方法的相关使用代码
   **/
        Debug.Log("以下是GetComponentsInChildren的相关使用代码。\nGetComponentsInChildren方
            法用来获取当前GameObject的子类中符合Type类型的所有组件。");

        //GetComponentsInChildren(type: Type, includeInactive: boolean = false)
        //返回GameObject自身及其所有子类对象中所有符合type类型的Component集合
        cjs = GetComponentsInChildren(typeof(ConfigurableJoint), false) as Component[];
        foreach (ConfigurableJoint cj in cjs)
        {
            Debug.Log("使用GetComponentsInChildren(type: Type, false)获取ConfigurableJoint：
                " + cj.name);
        }

        cjs = GetComponentsInChildren(typeof(ConfigurableJoint), true) as Component[];
        foreach (ConfigurableJoint cj in cjs)
        {
            Debug.Log("使用GetComponentsInChildren(type: Type, true)获取ConfigurableJoint："
                + cj.name);
        }

        //GetComponentsInChildren.<T> (includeInactive : boolean)
        cjs = GetComponentsInChildren<ConfigurableJoint>(true);
        foreach (ConfigurableJoint cj in cjs)
```

```
        {
            Debug.Log("使用GetComponentsInChildren.<T>(includeInactive : boolean)获取
                ConfigurableJoint：" + cj.name);
        }

        //GetComponentsInChildren.<T> ()
        cjs = GetComponentsInChildren<ConfigurableJoint>();
        foreach (ConfigurableJoint cj in cjs)
        {
            Debug.Log("使用GetComponentsInChildren.<T>()获取ConfigurableJoint：" + cj.name);
        }
    }
}
```


在这段代码中，首先分别演示了方法GetComponent (type:Type)、GetComponent.<T>()和GetComponent (type : String)的使用方法，并打印出了相应Component的InstanceID，如图3-4所示。然后分别演示了方法GetComponentInChildren(type:Type)和GetComponentInChildren.<T> ()的使用方法，并打印出了相应Component的name，如图3-5所示。接下来又分别演示了方法GetComponents(type:Type)和GetComponents.<T> ()的使用方法，并打印出了相应Component的InstanceID，如图3-6所示。最后分别演示了方法GetComponentsInChildren(type: Type, includeInactive: boolean = false)、GetComponentsInChildren.<T> (includeInactive : boolean)和GetComponentsInChildren.<T> ()的使用方法，并打印出了相应Component的name，如图3-7所示，图中右边数字为连续输出相同内容的次数。



图3-4 GetComponent的运行结果

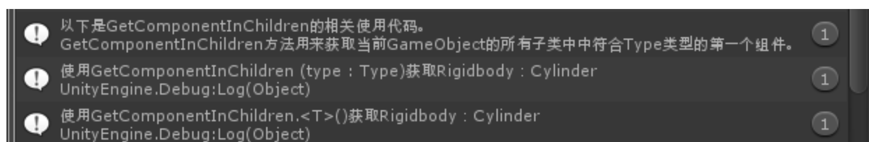


图3-5 GetComponentInChildren的运行结果



图3-6　GetComponents的运行结果

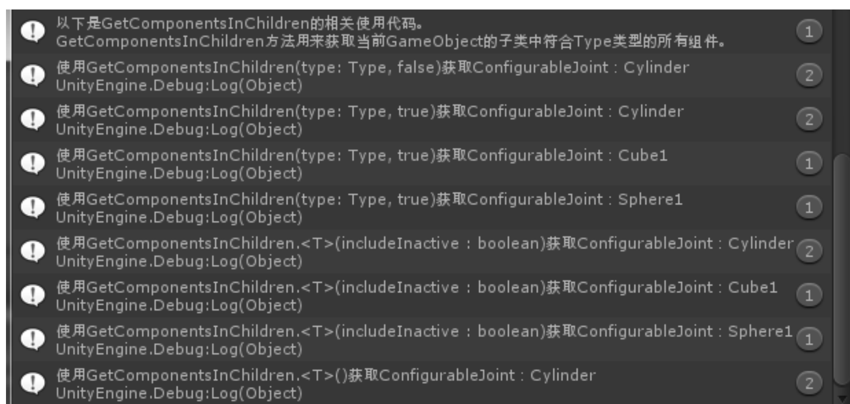


图3-7　GetComponentsInChildren的运行结果

3.3.2　SendMessage方法：发送消息

基本语法　(1) `public void SendMessage(string methodName);`

(2) `public void SendMessage(string methodName, object value);`

(3) `public void SendMessage(string methodName, SendMessageOptions options);`

(4) `public void SendMessage(string methodName, object value, SendMessageOptions options);`

上述4个重载方法涉及的参数有：参数methodName为接受信息的方法名字，参数value为信息的内容，参数options为信息接收的方式，默认为SendMessageOptions.RequireReceiver。

功能说明　此方法的功能是向GameObject自身发送消息，对其作用范围说明如下。

- 和自身同级的物体不会收到消息，例如，cube1和cube2的上一级父类都是cube0，则cube2不会收到cube1发送的消息。
- SendMessageOptions有两个可选方式：SendMessageOptions.RequireReceiver和SendMessageOptions.DontRequireReceiver。前者要求信息的接收方必须有接受信息的

方法，否则程序会报错，后者则无此要求。

提　示　与此方法功能相似的方法有BroadcastMessage和SendMessageUpwards，对其功能说明如下。

- BroadcastMessage 方法的功能是向自身及其所有子类发送消息。和自身同级的物体不会收到消息，例如，cube1 和 cube2 的上一级父类都是 cube0，则 cube2 不会收到 cube1 发送的消息。
- SendMessageUpwards 方法的功能是向 GameObject 自身及其所有父类发送消息。和自身同级的物体不会收到消息，例如，cube1 和 cube2 的上一级父类都是 cube0，则 cube2 不会收到 cube1 发送的消息。

实例演示　下面通过实例演示SendMessage相关方法的使用。在本实例工程中，有3个GameObject对象Cube1、Cube2和Cube3，它们的层级关系如图3-8所示，在Cube1、Cube2和Cube3中分别绑定有脚本BroadcastMessage_ts.cs、SendMessage_ts.cs和SendMessageUpward_ts.cs，它们的脚本内容很相似，此处以SendMessage_ts.cs脚本为例说明。

```
using UnityEngine;
using System.Collections;

public class SendMessage_ts : MonoBehaviour {
    void Start () {
        //向子类及自己发送信息
        //gameObject.BroadcastMessage("GetParentMessage",gameObject.name+":use
              BroadcastMessage send!");
        //向自己发送信息
        gameObject.SendMessage("GetSelfMessage",gameObject.name+":use SendMessage send!");
        ////向父类及自己发送信息
        //gameObject.SendMessageUpwards("GetChildrenMessage",gameObject.name+":use
              SendMessageUpwards send!");
    }
    //一个接受父类发送信息的方法
    private void GetParentMessage(string str){
        Debug.Log(gameObject.name + "收到父类发送的消息：" + str);
    }
    //一个接受自身发送信息的方法
    private void GetSelfMessage(string str)
    {
        Debug.Log(gameObject.name + "收到自身发送的消息：" + str);
    }
    //一个接受子类发送信息的方法
    private void GetChildrenMessage(string str)
    {
        Debug.Log(gameObject.name + "收到子类发送的消息：" + str);
    }
}
```

在这段代码中有3个方法GetParentMessage、GetSelfMessage和GetChildrenMessage，分别用来接受父类消息、自身消息和子类消息，并将消息打印出来，如图3-9所示。

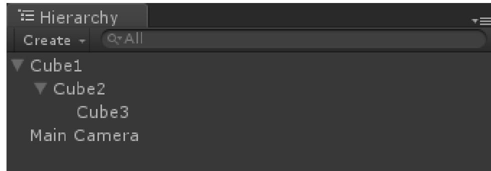


图3-8　实例工程中物体间层级关系

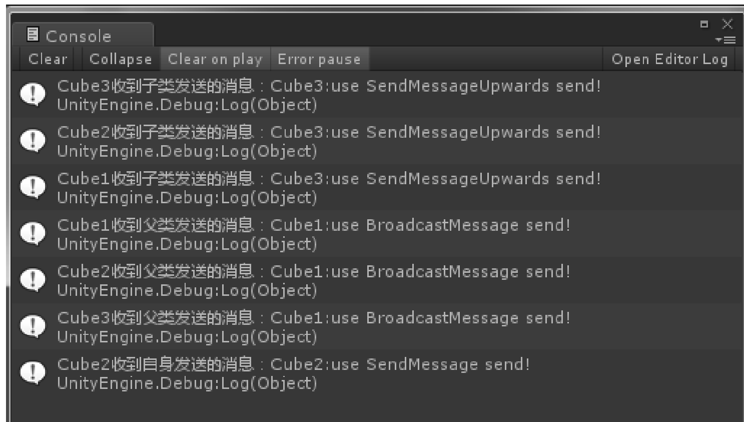


图3-9　SendMessage实例演示的运行结果

3.4　GameObject 类静态方法

在GameObject类中，涉及的静态方法主要有CreatePrimitive。在使用该方法创建GameObject对象时，往往会涉及添加组件（AddComponent）和查找对象（Find），因此此处将添加组件和查找对象的相关方法放到CreatePrimitive方法中介绍。

CreatePrimitive方法：创建GameObject对象

基本语法　public static GameObject CreatePrimitive(PrimitiveType type);

其中参数type为PrimitiveType的类型值。

功能说明　此方法的功能是创建一个系统自带的GameObject对象。

实例演示　下面通过实例演示CreatePrimitive方法以及Find类相关方法的使用。

```
using UnityEngine;
using System.Collections;

public class CreatePrimitive_ts : MonoBehaviour
{
    void Start()
```

```
    {
        //使用GameObject.CreatePrimitive方法创建GameObject
        GameObject g1 = GameObject.CreatePrimitive(PrimitiveType.Sphere);
        g1.name = "G1";
        g1.tag = "sphere_Tag";
        //使用 AddComponent (className : String)方法添加组件
        g1.AddComponent("SpringJoint");
        //使用AddComponent (componentType : Type)方法添加组件
        g1.AddComponent(typeof(GUITexture));
        g1.transform.position = Vector3.zero;

        GameObject g2 = GameObject.CreatePrimitive(PrimitiveType.Sphere);
        g2.name = "G2";
        g2.tag = "sphere_Tag";
        //使用AddComponent.<T>()方法添加组件
        g2.AddComponent<Rigidbody>();
        g2.transform.position = 2.0f * Vector3.right;
        g2.GetComponent<Rigidbody>().useGravity = false;

        GameObject g3 = GameObject.CreatePrimitive(PrimitiveType.Sphere);
        g3.name = "G1";
        g3.tag = "sphere_Tag";
        g3.transform.position = 4.0f * Vector3.right;

        //使用GameObject.Find类方法获取GameObject，返回符合条件的第一个对象
        Debug.Log("use Find:" + GameObject.Find("G1").transform.position);
        //使用GameObject.FindGameObjectWithTag类方法获取GameObject，返回符合条件的第一个对象
        Debug.Log("use FindGameObjectWithTag:" + GameObject.FindGameObjectWithTag("sphere_
            Tag").transform.position);
        //使用GameObject.FindGameObjectsWithTag类方法获取GameObject，返回符合条件的所有对象
        GameObject[] gos = GameObject.FindGameObjectsWithTag("sphere_Tag");
              foreach (GameObject go in gos)
        {
            Debug.Log("use FindGameObjectsWithTag:" + go.name + ":" + go.transform.position);
        }
        //更改g1、g2和g3的层级关系
        g3.transform.parent = g2.transform;
        g2.transform.parent = g1.transform;

        Debug.Log("use Find again1:" + GameObject.Find("G1").transform.position);
        //使用带"/"限定条件的方式查找GameObject
        //此处返回的对象需其父类为G2，且G2的父类名为G1
        //注意与上面不带"/"限定条件返回的对象的区别
        Debug.Log("use Find again2:" + GameObject.Find("/G1/G2/G1").transform.position);
    }
}
```

在这段代码中，首先用CreatePrimitive方法创建了3个不同的GameObject对象g1、g2和g3，并为它们设置了一些属性和不同的Component组件。然后分别使用方法Find、FindGameObjectWithTag和FindGameObjectsWithTag查找符合要求的对象，并打印出相应的信息，如图3-10所示。最后演示了使用“/”作为路径限定符来查找对象的方法。

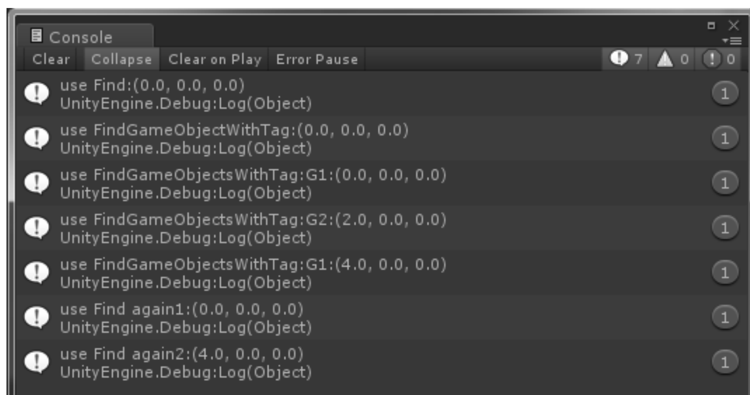


图3-10　CreatePrimitive实例演示的运行结果

3.5　关于 GameObject 类和 Component 类的使用注解

GameObject和Component是Unity中常用且非常重要的两个类，二者的实例属性和实例方法相似，只是在使用方法上稍有区别，所以本书不再对Component类作单独介绍，下面对这两个类之间的关系及其实例方法的使用进行简要说明。

- 通常一个GameObject对象由多个Component组成，而且一个GameObject对象至少有一个Transform组件。GameObject用来管理工程中的各个物体，而Component用来扩展这些物体自身的功能。
- GameObject类和Component类的属性名称和方法名称基本相同,各个属性和方法的用法也很相近，但它们仍有一些差别，以下以GetComponent方法为例说明。

若要获取当前脚本所在GameObject对象中的某个组件，直接使用GetComponent方法即可，如Rigidbody br= GetComponent<Rigidbody>()。若要获取非当前脚本所在GameObject对象中的某个组件，则需要有GameObject作为前置引用，如变量go为指向GameObject对象的引用，则：Rigidbody br= go.GetComponent<Rigidbody>()。

实例演示　下面通过实例演示GameObject类和Component类中GetComponent相关方法的使用。

```
using UnityEngine;
using System.Collections;

public class GameObjectAndComponent_ts : MonoBehaviour {
    public GameObject sp;
    void Start () {
        //以下3种表达方式功能一样，都返回当前脚本所在GameObject对象中的Rigidbody组件
        Rigidbody rb1 = rigidbody;
        Rigidbody rb2 = GetComponent<Rigidbody>();
        Rigidbody rb3 = rigidbody.GetComponent<Rigidbody>();
        Debug.Log("rb1的InstanceID：" + rb1.GetInstanceID());
```

```
        Debug.Log("rb2的InstanceID: " + rb2.GetInstanceID());
        Debug.Log("rb3的InstanceID: " + rb3.GetInstanceID());

        //使用前置引用获取引用对象的Rigidbody组件
        Debug.Log("前置引用sp对象中Rigidbody的InstanceID: "+sp.GetComponent<Rigidbody>().
            GetInstanceID());
    }
}
```


在这段代码中，首先声明了一个GameObject变量sp，用来指向外部GameObject对象，然后在Start方法中用3种不同的方法来获取当前脚本所在GameObject对象中的Rigidbody组件，并打印出它们的InstanceID，如图3-11所示。最后演示了使用GameObject前置引用来获取外部对象的Rigidbody组件的方法。由打印结果可知，虽然rb1、rb2和rb3的获取方式不一样，但都指向了相同的对象。

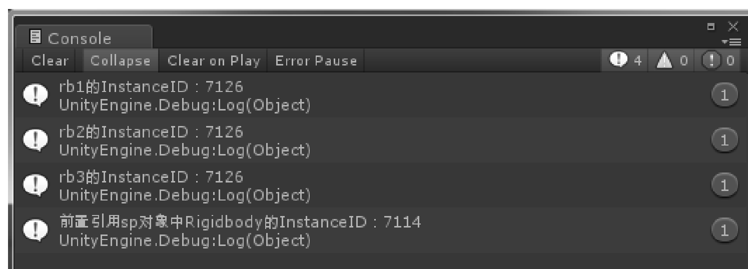


图3-11 本实例演示运行结果

第 4 章

HideFlags类

4

HideFlags为枚举类，用于控制Object对象的销毁方式及其在检视面板中的可视性。本章将对HideFlags类枚举成员的功能及使用方法进行较为详细的说明。

4.1 HideFlags 类枚举成员

枚举类HideFlags涉及的枚举成员有DontSave、HideAndDontSave、HideInHierarchy、HideInInspector、None和NotEditable，下面详细介绍这些枚举成员。

4.1.1 DontSave：保留对象到新场景

功能说明 此属性的功能是用来设置是否将Object对象保留到新的场景中，如果使用HideFlags.DontSave，则Object对象将在新场景中被保留下来，对其使用说明如下。

- 如果GameObject对象被HideFlags.DontSave标识，则在新Scene中GameObject的所有组件将被保留下来，但其子类GameObject对象不会被保留到新Scene中。
- 不可以对GameObject对象的某个组件如Transform进行HideFlags.DontSave标识，否则无效。
- 即使程序已经退出，被HideFlags.DontSave标识的对象也会一直存在于程序中，造成内存泄漏，对HideFlags.DontSave标识的对象，在不需要或程序退出时需要使用DestroyImmediate手动销毁。

实例演示 下面通过实例演示属性DontSave的使用。

本实例工程包含两个场景，下面是场景DontSave_unity中的脚本代码：

```
using UnityEngine;
using System.Collections;

public class DontSave_ts : MonoBehaviour {
    public GameObject go;
    public Transform t;
    void Start()
    {
```

```
        //GameObject对象使用HideFlags.DontSave可以在新scene中被保留
        go.hideFlags = HideFlags.DontSave;
        GameObject Pl = GameObject.CreatePrimitive(PrimitiveType.Plane);
        Pl.hideFlags = HideFlags.DontSave;
        //不可以对GameObject的组件设置HideFlags.DontSave，否则无效
        Transform tf = Instantiate(t, go.transform.position + new Vector3(2.0f, 0.0f, 0.0f),
            Quaternion.identity) as Transform;
        tf.hideFlags = HideFlags.DontSave;
        //导入名为newScene_unity的新scene
        Application.LoadLevel("newScene2_unity");
    }
}
```

4

在这段代码中，分别对场景中GameObject对象go、新创建的GameObject对象Pl和新实例化的Transform实例tf的hideFlags属性设置为HideFlags.DontSave，然后导入名为newScene2_unity的新场景。

下面是场景newScene2_unity中的脚本代码：

```
using UnityEngine;
using System.Collections;

public class NewScene2_ts : MonoBehaviour {
    GameObject cube, plane;
  void Start () {
        Debug.Log("这是NewScene2！");
  }
    //当程序退出时用DestroyImmediate()销毁被HideFlags.DontSave标识的对象
    //否则即使程序已经退出，被HideFlags.DontSave标识的对象依然在Hierarchy面板中
    //即每运行一次程序就会产生多余对象，造成内存泄漏
    void OnApplicationQuit()
    {
        cube = GameObject.Find("Cube0");
        plane = GameObject.Find("Plane");
        if (cube)
        {
            Debug.Log("Cube0 DestroyImmediate");
            DestroyImmediate(cube);
        }
        if (plane)
        {
            Debug.Log("Plane DestroyImmediate");
            DestroyImmediate(plane);
        }
    }
}
```

在这段脚本中，首先声明了两个GameObject类型的变量cube和plane，然后在OnApplicationQuit()方法中查找当前场景中是否存在名为Cube0和Plane的GameObject对象，如果存在，则在程序退出时将它们立即销毁。图4-1为程序启动前场景中的物体，图4-2为程序启动后场景中的物体，图4-3为在程序退出时未将场景中被保留的物

体销毁时的结果(即把场景newScene2_unity脚本中OnApplicationQuit()中的代码注释掉后程序的运行结果)。如果在程序退出时销毁了被ideFlags.DontSave标识的对象，则程序退出后状态如图4-1所示。

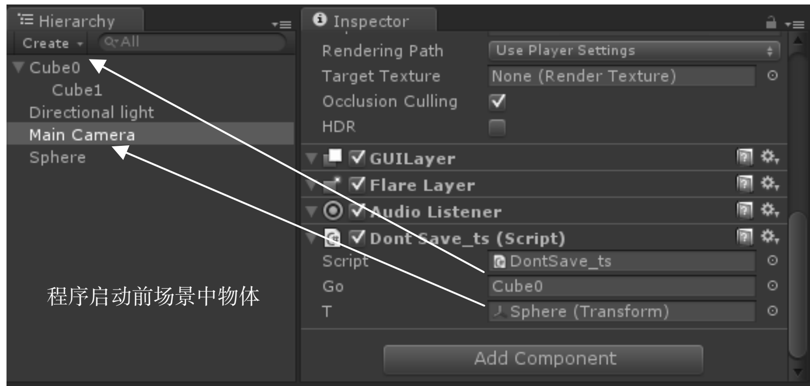


图4-1　程序启动前场景中的物体

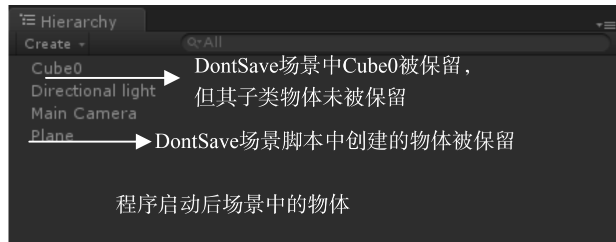


图4-2　程序启动后场景中的物体



图4-3　程序退出后场景中的物体

4.1.2　HideAndDontSave：保留对象到新场景

功能说明　此属性的功能是用来设置是否将Object对象保留到新Scene中，如果使用HideFlags.HideAndDontSave，则Object对象将在新Scene中被保留下来，但不会显示在Hierarchy面板中。属性HideAndDontSave与DontSave的功能类似，请参考属性DontSave的功能说明。

实例演示 请参考属性DontSave的实例演示，只需把DontSave脚本里的HideFlags. DontSave改成HideFlags.HideAndDontSave，即可看到两者的不同。

4.1.3 HideInHierarchy：在Hierarchy面板中隐藏

功能说明 此属性的功能是设置Object对象在Hierarchy面板中是否被隐藏，对其使用说明如下。

- 若要在Hierarchy面板中隐藏不是在脚本中创建的对象，需要在Awake方法中使用HideFlags.HideInHierarchy才能生效。
- 若隐藏父物体，子物体也会被隐藏掉，但隐藏子物体，父物体不会被影响。

实例演示 下面通过实例演示属性HideInHierarchy的使用。

```
using UnityEngine;
using System.Collections;

public class HideInHierarchy_ts : MonoBehaviour
{
    public GameObject go, sub;
    public Transform t;
    void Awake()
    {
        //go、sub、gameObject为已存在对象，需在Awake方法中使用HideFlags.HideInHierarchy
        go.hideFlags = HideFlags.HideInHierarchy;
        sub.hideFlags = HideFlags.HideInHierarchy;
        gameObject.hideFlags = HideFlags.HideInHierarchy;
    }

    void Start()
    {
        //Pl、tf是在代码中创建的对象，可以在任何方法中使用HideFlags.HideInHierarchy
        GameObject Pl = GameObject.CreatePrimitive(PrimitiveType.Plane);
        Pl.hideFlags = HideFlags.HideInHierarchy;
        Transform tf = Instantiate(t, go.transform.position + new Vector3(2, 0.0f, 0.0f),
            Quaternion.identity) as Transform;
        tf.hideFlags = HideFlags.HideInHierarchy;
    }
}
```

在这段代码中，首先声明了两个GameObject类型的变量go和sub，然后在Awake方法中设置其hideFlags属性为HideFlags.HideInHierarchy。然后在Start方法中临时创建和实例化了两个GameObject对象Pl和tf，并对它们的hideFlags进行设置。图4-4是程序启动前Hierarchy面板的状况，图4-5是程序启动后Hierarchy面板的状况，从运行结果可以发现，原来的cube对象包括其子类Sphere、cube1、MainCamera及新创建的两个对象Pl和tf都被隐藏了。

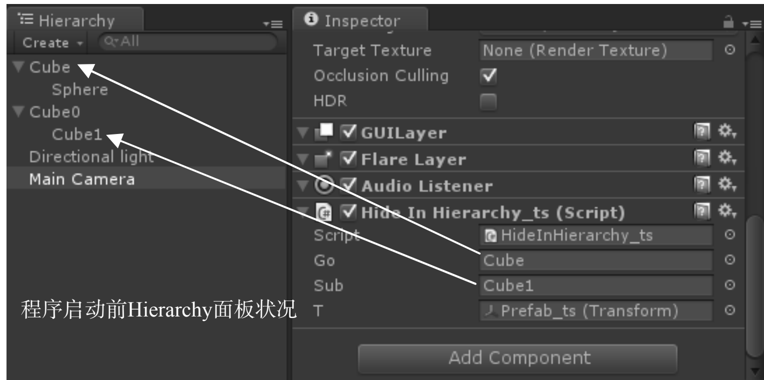


图4-4　程序启动前Hierarchy面板状况

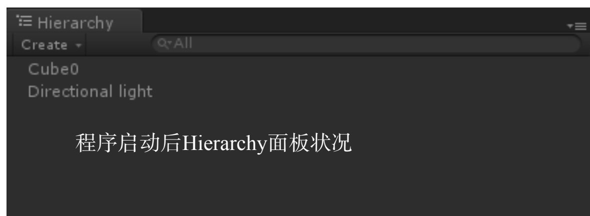


图4-5　程序启动后Hierarchy面板状况

4.1.4　HideInInspector：在Inspector面板中隐藏

功能说明　此属性的功能是设置Object对象在Inspector面板中是否被隐藏，对其使用说明如下。

- 如果一个GameObject使用了HideFlags.HideInInspector，则其所有组件将在Inspector面板中被隐藏，但其子类对象的组件仍可在Inspector面板中显示。
- 如果只隐藏了GameObject对象的某个组件，如Transform，则并不影响GameObject的其他组件如Renderer、Rigidbody等在Inspector面板中的显示状态。

实例演示　下面通过实例演示属性HideInInspector的使用。

```
using UnityEngine;
using System.Collections;

public class HideInInspector_td : MonoBehaviour {
    public GameObject go;
    public Transform t;
    void Start()
    {
        //go、gameObject、Pl都是GameObject对象，使用HideFlags.HideInInspector后
        //其所有组件将在Inspector面板中隐藏
        //但并不影响其子类在Inspector面板中的显示
        go.hideFlags = HideFlags.HideInInspector;
        gameObject.hideFlags = HideFlags.HideInInspector;
```

```
            GameObject Pl = GameObject.CreatePrimitive(PrimitiveType.Plane);
            Pl.hideFlags =  HideFlags.HideInInspector;
            //tf为Transform对象，使用HideFlags.HideInInspector后
            //tf对应的GameObject的Transform组件将在Inspector面板中隐藏
            //但GameObject的其他组件仍可在Inspector面板中显示
            Transform tf = Instantiate(t, go.transform.position + new Vector3(2.0f, 0.0f, 0.0f),
                Quaternion.identity) as Transform;
            tf.hideFlags = HideFlags.HideInInspector;
    }
}
```

在这段代码中，首先声明了一个GameObject类型变量go，然后在Start方法中对go及MainCamera的hideFlags设置为HideFlags.HideInInspector，接着分别创建和实例化了一个GameObject对象Pl和一个Transform实例tf，并将它们的hideFlags都设置为HideFlags.HideInInspector。运行程序可以发现，对象go、MainCamera及Pl的Inspector面板中的组件都被隐藏，tf组件也被隐藏，但其所在GameObject对象的其他组件却未被隐藏，请自行运行程序查看。

4

4.1.5 None：HideFlags默认值

功能说明 None为HideFlags的默认值，即不改变Object对象的可见性。

4.1.6 NotEditable：对象在Inspector面板中的可编辑性

功能说明 此属性用来设置在程序运行时Object对象是否可在Inspector面板中被编辑，对其使用说明如下。

- GameObject对象使用HideFlags.NotEditable可以使得GameObject对象的所有组件在Inspector面板中都处于不可编辑状态。但GameObject对象被HideFlags.NotEditable标识并不影响其子类对象的可编辑性。
- 对于GameObject对象的某个组件如Transform单独使用HideFlags.NotEditable，只会使得当前组件不可编辑，但GameObject的其他组件仍可在Inspector面板中编辑。

实例演示 下面通过实例演示属性NotEditable的使用。

```
using UnityEngine;
using System.Collections;

public class NotEditable_ts : MonoBehaviour {
    public GameObject go;
    public Transform t;
    void Start()
    {
        //GameObject对象使用HideFlags.NotEditable可以使得GameObject的
        //所有组件在Inspector面板中都处于不可编辑状态
        //GameObject对象被HideFlags.NotEditable标识并不影响其子类的可编辑性
        go.hideFlags = HideFlags.NotEditable;
```

```
        GameObject Pl = GameObject.CreatePrimitive(PrimitiveType.Plane);
        Pl.hideFlags = HideFlags.NotEditable;

        //对于GameObject的某个组件单独使用HideFlags.NotEditable
        //只会使得当前组件不可编辑，但GameObject的其他组件仍可编辑
        t.hideFlags = HideFlags.NotEditable;
        Transform tf = Instantiate(t, go.transform.position + new Vector3(2.0f, 0.0f, 0.0f),
            Quaternion.identity) as Transform;
        tf.hideFlags = HideFlags.NotEditable;
    }
}
```

在这段代码中，首先声明了一个GameObject类型变量go，然后在Start方法中对go、新创建的GameObject对象Pl及新实例化的Transform实例tf的hideFlags属性设置为HideFlags. NotEditable。运行程序可以发现，对象go和Pl的所有组件将不可被编辑，组件tf也不可被编辑，但组件tf所在的GameObject对象的其他组件仍可被编辑，请自行运行程序查看。

4.2 HideFlags 类使用小结

本章对HideFlags类的枚举成员进行了较为详细的介绍。在HideFlags类的6个枚举成员中，用得较多的是DontSave，使用它将Object对象保留到新Scene中时，需要注意在合适的时机将Object对象手动销毁，以防出错。若想将场景中的某个Object对象在Hierarchy面板或Inspector面板中隐藏，可以使用成员HideInHierarchy或HideInInspector，但要注意对象间的层次问题，具体请查看它们各自的功能说明。

Mathf类

Mathf类是Unity中的数学类，属于结构体类型，只有静态属性和静态方法，即不可实例化。在使用时，直接调用其静态属性或静态方法，如Mathf.PI、Mathf.Sin(1)等。本章主要介绍了Mathf类的一些静态属性和静态方法。

5.1 Mathf 类静态属性

在Mathf类中，涉及的静态属性有Deg2Rad、Rad2Deg和Infinity，其中属性Deg2Rad和Rad2Deg功能相似，因此放到一起介绍，下面详细介绍这些属性。

5.1.1 Deg2Rad属性：从角度到弧度常量

基本语法 `public const float Deg2Rad = 0.0174533f;`

功能说明 此属性用来表示数学计算中从角度到弧度转变的常量值，其值为(2*Mathf.PI)/360=0.01745329，此属性只读。

提　　示 Rad2Deg属性与此属性功能相反，是从弧度到角度的转换常量，其值为57.2958f。

实例演示 下面通过实例打印出属性Deg2Rad和Rad2Deg的值。

```
using UnityEngine;
using System.Collections;

public class DegAndRad_ts : MonoBehaviour {
    void Start () {
        //从角度到弧度转换常量
        Debug.Log("Mathf.Deg2Rad:" + Mathf.Deg2Rad);
        //从弧度到角度转换常量
        Debug.Log("Mathf.Rad2Deg:" + Mathf.Rad2Deg);
    }
}
```

在这段代码中，分别打印出了Mathf.Deg2Rad和Mathf.Rad2Deg的值，结果如图5-1所示。

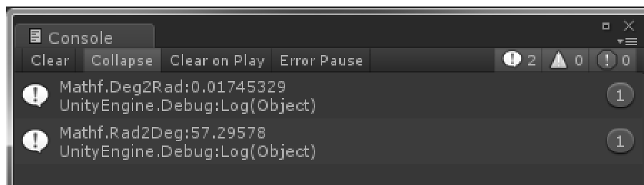


图5-1　实例演示的运行结果

5.1.2　Infinity属性：正无穷大

基本语法　`public const float Infinity = 1.0f / 0.0f;`

功能说明　此属性用来表示在数学计算中的正无穷大，只读。其计算规则及使用说明如下。

- ❑ Mathf.Infinity ÷ x=Mathf.Infinity，其中x为一个具体数值，如10000。
- ❑ Mathf.Infinity ÷ Mathf.Infinity=NaN，即计算结果不是数值（Not a Number）。
- ❑ Mathf.Infinity只是在Unity中的一个正无穷大数值的表示，不代表任何具体数值，不要把其用在具体的数值计算中。

实例演示　下面通过实例演示属性Infinity的使用。

```
using UnityEngine;
using System.Collections;

public class Infinity_ts : MonoBehaviour
{
    void Start()
    {
        Debug.Log("0:" + Mathf.Infinity);
        Debug.Log("1:" + Mathf.Infinity / 10000.0f);
        Debug.Log("2:" + Mathf.Infinity / Mathf.Infinity);
    }
}
```

在这段代码中，分别打印出了Infinity的值、Infinity与具体数值相除后的值以及Infinity与自身相除后的值，结果如图5-2所示，可以发现Infinity与具体数值相除后仍为Infinity，与自身相除会得到一个非数值的结果（NaN:Not a Number）。

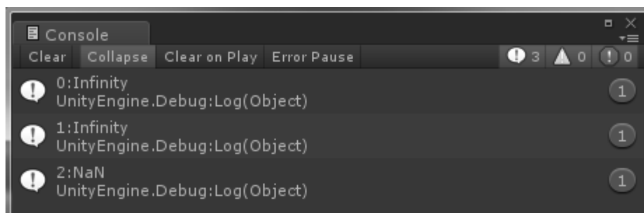


图5-2　Infinity实例演示的运行结果

5.2 Mathf 类静态方法

在Mathf类中，涉及的静态方法有Clamp方法、ClosestPowerOfTwo方法、DeltaAngle方法、InverseLerp方法、Lerp方法、LerpAngle方法、MoveTowards方法、MoveTowardsAngle方法、PingPong方法、Repeat方法、Round方法、SmoothDamp方法、SmoothDampAngle方法和SmoothStep方法，下面详细介绍这些方法。

5.2.1 Clamp方法：返回有限范围值

基本语法 (1) public static float Clamp(float value, float min, float max);

其中参数min为返回值的最小值，参数max为返回值的最大值。参数和返回值类型为浮点型。

(2) public static int Clamp(int value, int min, int max);

其中参数min为返回值的最小值，参数max为返回值的最大值。参数和返回值类型为整型。

功能说明 此方法用来返回有范围限制的value值，当value∈[min, max]时返回value值；当value<min时返回min值；当value>max时返回max值。

提　示 方法Clamp01与此方法功能相似，不同之处是，Clamp01的取值范围为[0,1]，只有一个参数。当参数value∈[0,1]时返回value值；当参数value<0时返回0值；当参数value>1时返回值为1。

实例演示 下面通过实例演示方法Clamp的使用。

```
using UnityEngine;
using System.Collections;

public class Clamp_ts : MonoBehaviour
{
    void Start()
    {
        Debug.Log("当value<min时：" + Mathf.Clamp(-1, 0, 10));
        Debug.Log("当min<value<max时：" + Mathf.Clamp(3, 0, 10));
        Debug.Log("当value>max时：" + Mathf.Clamp(11, 0, 10));
        //方法Clamp01的返回值范围为[0,1]
        Debug.Log("当value<0时:" + Mathf.Clamp01(-0.1f));
        Debug.Log("当0<value<1时:" + Mathf.Clamp01(0.5f));
        Debug.Log("当value>1时:" + Mathf.Clamp01(1.1f));
    }
}
```

在这段代码的Start方法中，分别打印出了方法Clamp和Clamp01在不同情况下的返回值，输出结果如图5-3所示。

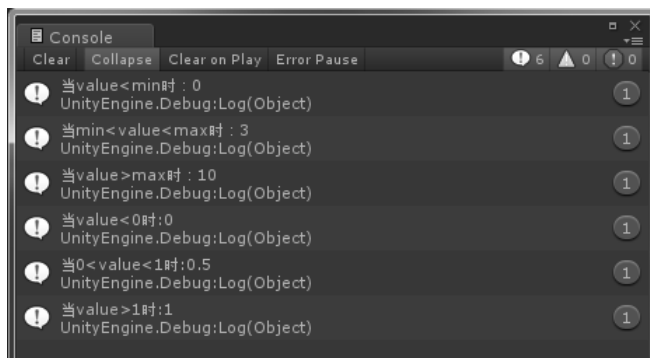


图5-3　方法Clamp实例演示结果输出

5.2.2　ClosestPowerOfTwo方法：返回 2 的某次幂

基本语法　public static int ClosestPowerOfTwo(int value);

功能说明　此方法用于返回最接近参数值value的2的某次幂的值。当value属于中间值时取较大值，例如：

f=Mathf. ClosestPowerOfTwo(11)，则f=8；

f=Mathf. ClosestPowerOfTwo(12)，则f=16；

当value值小于0时，返回值为0。

实例演示　下面通过实例演示方法ClosestPowerOfTwo的使用。

```
using UnityEngine;
using System.Collections;

public class ClosestPowerOfTwo_ts : MonoBehaviour
{
    void Start()
    {
        Debug.Log("11与8最接近，输出值为：" + Mathf.ClosestPowerOfTwo(11));
        Debug.Log("12与8和16的差值都为4，输出值为：" + Mathf.ClosestPowerOfTwo(12));
        Debug.Log("当value<0时，输出值为：" + Mathf.ClosestPowerOfTwo(-1));
    }
}
```

在这段代码的Start方法中，分别测试了3种不同情况下方法ClosestPowerOfTwo的返回值，输出结果如图5-4所示。

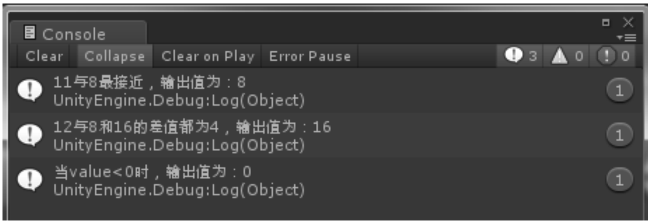


图5-4 方法ClosestPowerOfTwo实例演示输出结果

5.2.3 DeltaAngle方法：最小增量角度

基本语法 public static float DeltaAngle(float current, float target);

其中参数current为当前角度，参数target为目标角度。

功能说明 此方法用于返回从参数值current到target的最小增量角度值。计算方法为，首先将current和target按照360度为一周换算到区间（-180,180]中，设current和target换算后的值分别对应c和t，它们在坐标轴中的夹角为e（0≤e≤180），则若c经过顺时针旋转e度能到达t，则返回值为e，如图5-5所示；若c经过逆时针旋转e度能到达t，则返回值为-e，如图5-6所示。

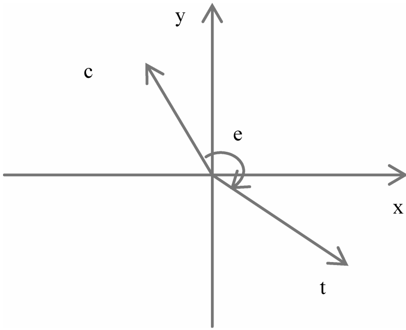


图5-5 顺时针换算

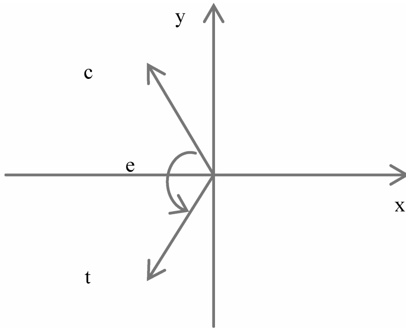


图5-6 逆时针换算

实例演示 下面通过实例验证方法DeltaAngle的算法。

```
using UnityEngine;
using System.Collections;

public class DeltaAngle_ts : MonoBehaviour {
    void Start () {
        //1180=360*3+100,即求100和90之间的夹角
        Debug.Log(Mathf.DeltaAngle(1180, 90));
        //-1130=-360*3-50,即求-50和90之间的夹角
        Debug.Log(Mathf.DeltaAngle(-1130, 90));
        //-1200=-360*3-120,即求-120和90之间的夹角
        Debug.Log(Mathf.DeltaAngle(-1200, 90));
```

```
    }
}
```

在这段代码中，分别检验了3组不同数据的计算结果，每组数据结果的计算方法如代码中注释所述，程序运行结果如图5-7所示。

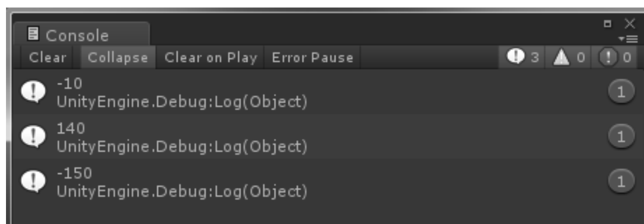


图5-7　DeltaAngle实例演示的运行结果

5.2.4　InverseLerp方法：计算比例值

基本语法　public static float InverseLerp(float from, float to, float value);

其中参数from为起始值，参数to为终点值，参数value为参考值。

功能说明　此方法用来返回value值在从参数from到to中的比例值。设f=Mathf.Inverse Lerp(from,to,v)，其中f、from、to和v都是float类型，$f'=\frac{v-from}{to-from}$，则若 $f'\in[0.0f,1.0f]$.，则$f=f'$；若$f'<0$，则$f=0$；若$f'>1$则$f=1$。

实例演示　下面通过实例验证方法InverseLerp的算法。

```
using UnityEngine;
using System.Collections;

public class InverseLerp_ts : MonoBehaviour {
    void Start () {
        float f,from,to,v;
        from=-10.0f;
        to=30.0f;
        v=10.0f;
        f = Mathf.InverseLerp(from,to,v);
        Debug.Log("当0<f'<1时： "+f);
        v = -20.0f;
        f = Mathf.InverseLerp(from, to, v);
        Debug.Log("当f'<0时： " + f);
        v = 40.0f;
        f = Mathf.InverseLerp(from, to, v);
        Debug.Log("当f'>1时： " + f);
    }
}
```

在这段代码中，首先声明了4个变量f、from、to和v，然后对v值分别赋予不同的值，

并调用方法InverseLerp，最后分别打印出返回值，运行结果如图5-8所示。

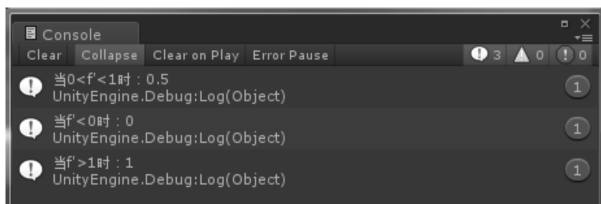


图5-8 InverseLerp实例演示的运行结果

5.2.5 Lerp方法：线性插值

基本语法 public static float Lerp(float from, float to, float t);

其中参数from为线性插值的起始值，参数to为线性插值的结束值，参数t为插值系数。

功能说明 此方法的功能是用来返回一个从from到to范围的线性插值。返回值的计算方法为(to-from)*t'+from，其中：

- 参数t的有效取值范围为[0,1]，当t<0时有效值t'=0，当t>1时有效值t'=1；
- 参数from和to为任意的float数值，from和to之间没有任何约束关系，from的值可以大于to也可以小于to。

实例演示 下面通过实例演示方法Lerp的使用。

```
using UnityEngine;
using System.Collections;

public class Lerp_ts : MonoBehaviour {
   float r, g, b;
   void FixedUpdate () {
       r = Mathf.Lerp(0.0f,1.0f,Time.time*0.2f);
       g = Mathf.Lerp(0.0f, 1.0f, -1.0f+Time.time * 0.2f);
       b = Mathf.Lerp(0.0f, 1.0f, -2.0f+Time.time * 0.2f);
       light.color = new Color(r, g, b);
   }
}
```

在这段代码中，首先声明了3个float类型的变量r、g和b，分别用来记录Color的RGB分量值。然后在FixedUpdate方法中使用方法Lerp使得r、g、b随着时间依次递增。运行程序可以发现，灯光（light）的颜色会由黑变红接着变黄最后变成白色，请自行运行程序查看。

5.2.6 LerpAngle方法：角度插值

基本语法 public static float LerpAngle(float a, float b, float t);

其中参数a为起始角度，参数b为结束角度，参数t为插值系数。

功能说明　此方法的功能是用来返回从角度a到b之间的一个插值。此方法功能与方法Lerp (from : float, to : float, t : float)类似，只是此方法主要用来对角度之间进行插值，其规则如下。

- a和b可以为任意的float类型数值，a可以大于b，也可小于b，它们之间没有任何约束关系。
- 插值系数t的有效取值范围为[0,1]，当t<0时其有效值t'=0，当t>1时其有效值t'=1。
- 插值计算之前需要先对a、b进行规范化，以确定需要插值的大小，对a、b规范化规则如下（以参数a为例，b与此相同）：

$$a'=360*k+a，其中k为整数，可求k的值使得a' \in [0，360];$$

设对a、b进行规范化后的值分别为a'、b'，它们角度值大小对应的二维坐标如图5-9所示。设a'和b'之间的差值为c，c可能是a'–b'的值，也可能是b'–a'的值，总之须使得c ∈ [0，180]。当a'沿着顺时针方向旋转c度与b'重合时（如图5-9中β所示），则插值计算方式为：f=a–c*t'，其中f即为方法的返回值，t'为t的有效值；

当a'沿着逆时针方向旋转c度与b'重合时（如图5-9中α所示），则插值计算方式为：f=a+c*t'，其中f即为方法的返回值，t'为t的有效值；

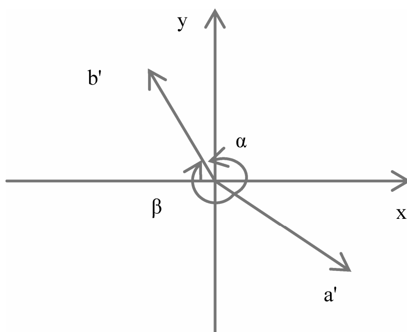


图5-9　角度规范化示意图

实例演示　下面通过实例验证方法LerpAngle的算法。

```
using UnityEngine;
using System.Collections;

public class LerpAngle_ts : MonoBehaviour {
    void Start () {
        float a, b;
        a = -50.0f;
        b = 400.0f;
        //a'=360-50=310,b'=-360+400=40,从而可知c=90
        //从a'沿着逆时针方向可经90度到b'，故返回值f=a+c*t=-50+90*0.3=-23
        Debug.Log("test1:"+Mathf.LerpAngle(a,b,0.3f));
```

```
            a = 400.0f;
            b = -50.0f;
            //a'=-360+400=40,b'=360-50=310,从而可知c=90
            //从a'沿着顺时针方向可经90度到b', 故返回值f=a-c*t=400-90*0.3=373
            Debug.Log("test2:" + Mathf.LerpAngle(a, b, 0.3f));
        }
    }
```

在这段代码中，首先声明了两个变量a和b，然后分别对a、b赋值，当a=–50.0f，b=400.0f时，对其规范化处理后a'=360–50=310，b'=–360+400=40，从a'沿着逆时针方向可经90度到b'，故返回值f=a+c*t=–50+90*0.3=–23；当a=400.0f，b=–50.0f时，对其规范化处理后a'=–360+400=40,b'=360–50=310，从a'沿着顺时针方向可经90度到b'，故返回值f=a–c*t=400–90*0.3=373，程序运行结果如图5-10所示。

5

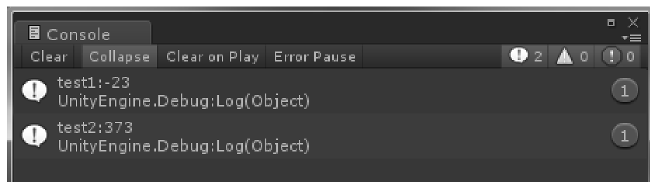


图5-10 LerpAngle实例演示的运行结果

5.2.7 MoveTowards方法：选择性插值

基本语法 public static float MoveTowards(float current, float target, float maxDelta);

其中参数current为当前值，参数target为目标值，参数maxDelta为最大约束值。

功能说明 此方法的功能是返回一个从current到target之间的插值，返回值受maxDelta值的约束。设a、b和d分别为3个float类型的数值，且方法MoveTowards(a,b,d)的返回值为c，则返回值c的计算方式如下。

- 若a<b：当a+d<b时，c=a+d；当a+d>b时，c=b；
- 若a>b：当a–d>b时，c=a–d；当a–d<b时，c=b。

实例演示 下面通过实例验证方法MoveTowards的算法。

```
using UnityEngine;
using System.Collections;
public class MoveTowards_ts : MonoBehaviour
{
    void Start()
    {
        float a, b, d;
        a = 10.0f;
        b = -10.0f;
        d = 5.0f;
        //a>b,且a-d>b, 返回值为a-d
        Debug.Log("Test01:" + Mathf.MoveTowards(a, b, d));
```

```
        d = 50.0f;
        //a>b,且a-d<b, 返回值为b
        Debug.Log("Test02:" + Mathf.MoveTowards(a, b, d));

        a = 10.0f;
        b = 50.0f;
        d = 5.0f;
        //a<b,且a+d<b, 返回值为a+d
        Debug.Log("Test03:" + Mathf.MoveTowards(a, b, d));
        d = 50.0f;
        //a<b,且a+d>b, 返回值为b
        Debug.Log("Test04: " + Mathf.MoveTowards(a, b, d));
    }
}
```

在这段代码中，首先声明了3个变量a、b和d，然后分别对其赋予不同的值，并打印出方法MoveTowards(a,b,d)的返回值，具体的算法检验请查看代码中注释，图5-11为程序运行的输出结果。

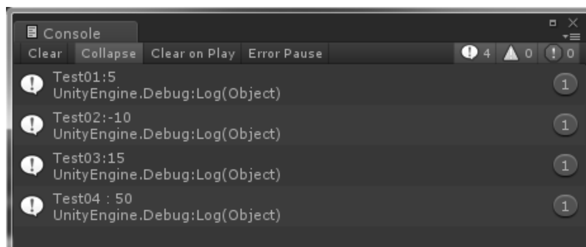


图5-11　MoveTowards实例演示的运行结果

5.2.8　MoveTowardsAngle方法：角度的选择性插值

基本语法　public static float MoveTowardsAngle(float current, float target, float maxDelta);

其中参数curren为起始角度，参数target为结束角度，参数maxDelta为最大约束值。

功能说明　此方法的作用是返回一个从当前角度current向目标角度target旋转的插值，每帧旋转角度不超过maxDelta度。此方法与方法MoveTowards方法功能类似，此方法主要用于角度的旋转变换，maxDelta值越大，旋转速度相对越快。

实例演示　下面通过实例演示方法MoveTowardsAngle的使用。

```
using UnityEngine;
using System.Collections;

public class MoveTowardsAngle_ts : MonoBehaviour
{
    float targets = 0.0f;
    float speed = 40.0f;
    void Update()
    {
```

```
        //每帧不超过speed * Time.deltaTime度
        float angle = Mathf.MoveTowardsAngle(transform.eulerAngles.y, targets, speed * Time.
            deltaTime);
        transform.eulerAngles = new Vector3(0, angle, 0);
    }
    void OnGUI()
    {
        if (GUI.Button(new Rect(10.0f, 10.0f, 200.0f, 45.0f), "顺时针旋转90度"))
        {
            targets += 90.0f;
        }
        if (GUI.Button(new Rect(10.0f, 60.0f, 200.0f, 45.0f), "逆时针旋转90度"))
        {
            targets -= 90.0f;
        }
    }
}
```

在这段代码中，首先声明两个变量targets和speed，变量targets用于记录物体旋转的目标角度，可在OnGUI方法中设置，变量speed用于控制物体每帧旋转的最大角度。然后在Update方法中调用方法MoveTowardsAngle，返回一个从物体当前角度到目标角度的一个插值。最后将这个插值赋给transform的eulerAngles。请自行运行程序查看。

5.2.9　PingPong方法：往复运动

基本语法　public static float PingPong(float t, float length);

功能说明　此方法用于模拟乒乓球的往复运动。设f＝Mathf.PingPong(t,l)，其中f、t和l均为float类型数值。

(1) 若l>0，则：

$$f=\begin{cases}|t|\%l, & 当|t|\%2l\leqslant l时;\\ l-|t|\%l, & 当|t|\%2l>l时;\end{cases}$$

(2) 若l<0，则：

$$f=\begin{cases}2l+|t|\%|2l|, & 当|t|\%|2l|\leqslant l时;\\ -|t|\%|2l|, & 当|t|\%|2l|>l时;\end{cases}$$

实例演示　下面通过实例验证方法PingPong的算法。

```
using UnityEngine;
using System.Collections;

public class PingPong_ts : MonoBehaviour {
    void Start () {
        float f, t, l;
```

```
            t = 11.0f;
            l = 5.0f;
            f = Mathf.PingPong(t,l);
            Debug.Log("l>0,|t|%2l<=l时: "+f);
            t = 17.0f;
            l = 5.0f;
            f = Mathf.PingPong(t, l);
            Debug.Log("l>0,|t|%2l>l时: " + f);
            t = 11.0f;
            l = -5.0f;
            f = Mathf.PingPong(t, l);
            Debug.Log("l<0,|t|%2l<=l时: " + f);
            t = 17.0f;
            l = -5.0f;
            f = Mathf.PingPong(t, l);
            Debug.Log("l<0,|t|%2l>l时: " + f);
        }
    }
```

在这段代码中，首先声明了3个变量f、t和l，然后对变量t和l分别赋予不同的值，最后调用方法PingPong，并打印其返回值，运行结果如图5-12所示。对输出结果的计算方法请参考功能说明。

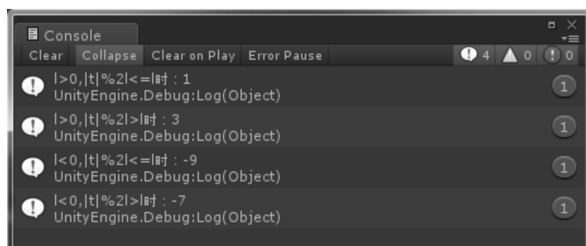


图5-12　PingPong实例演示的运行结果

5.2.10　Repeat方法：取模运算

基本语法　public static float Repeat(float t, float length);

功能说明　此方法的作用类似于浮点数的取模运算。设f = Mathf.Repeat(t,l)，其中f、t和l为float类型数值，则：

- 当t>0，l>0时，f=t%l；
- 当t<0，l<0时，f=–(–t%–l)；
- 当t>0，l<0时，f=l+t%–l；
- 当t<0，l>0时，f=–(l+(–t)%l)；
- 当l=0时，f=NaN，NaN=Not a Number。

实例演示　下面通过实例验证方法Repeat的算法。

```
using UnityEngine;
using System.Collections;

public class Repeat_ts : MonoBehaviour {
    void Start () {
        float f, t, l;
        t = 12.5f;
        l = 5.3f;
        f = Mathf.Repeat(t,l);
        Debug.Log("t>0,l>0时: "+f);
        t = -12.5f;
        l = -5.3f;
        f = Mathf.Repeat(t, l);
        Debug.Log("t<0,l<0时: " + f);
        t = 12.5f;
        l = -5.3f;
        f = Mathf.Repeat(t, l);
        Debug.Log("t>0,l<0时: " + f);
        t = -12.5f;
        l = 5.3f;
        f = Mathf.Repeat(t, l);
        Debug.Log("t<0,l>0时: " + f);
        t = -12.5f;
        l = 0.0f;
        f = Mathf.Repeat(t, l);
        Debug.Log("t<0,l==0时: " + f);
  }
}
```

在这段代码中，首先声明了3个变量f、t和l，然后对变量t和l分别赋予不同的值，最后调用方法Repeat，并打印出其返回值，运行结果如图5-13所示。对输出结果的计算方法请参考功能说明。

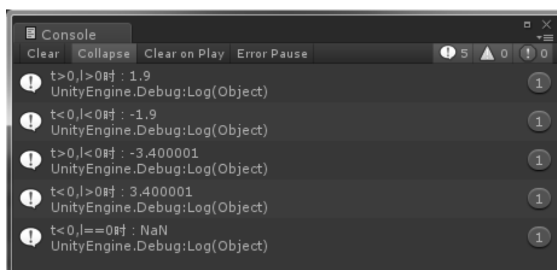


图5-13 Repeat实例演示的运行结果

5.2.11 Round方法：浮点数的整型值

基本语法 `public static float Round(float f);`

功能说明 此方法的作用是返回离f最近的整型浮点值。设a和b分别为浮点数f的整数部分和小数部分，即f=a+b，则Round(f)的计算规则如下。

- 当|b|< 0.5时，无论f为正数还是负数，Round(f)的返回值为a；
- 当|b|> 0.5时，若f为正数，则Round(f)返回值为a+1；若f为负数，则Round(f)返回值为a−1。
- 当|b|= 0.5时，若a为偶数，则Round(f)返回值为a；若a为奇数，则当f<0时Round(f)的返回值为a−1，当f>0时Round(f)的返回值为a+1。

提　　示　RoundToInt方法与此方法功能相同，只是RoundToInt方法的返回值类型为整型。

实例演示　下面通过实例演示方法Round的使用。

```
using UnityEngine;
using System.Collections;

public class Round_ts : MonoBehaviour
{
    void Start()
    {
        //设Round(f)中f=a.b
        Debug.Log("b<0.5,f>0：" + Mathf.Round(2.49f));
        Debug.Log("b<0.5,f<0：" + Mathf.Round(-2.49f));
        Debug.Log("b>0.5,f>0：" + Mathf.Round(2.61f));
        Debug.Log("b>0.5,f<0：" + Mathf.Round(-2.61f));
        Debug.Log("b=0.5,a为偶数,f>0：" + Mathf.Round(6.5f));
        Debug.Log("b=0.5,a为偶数,f<0：" + Mathf.Round(-6.5f));
        Debug.Log("b=0.5,a为奇数,f>0：" + Mathf.Round(7.5f));
        Debug.Log("b=0.5,a为奇数,f<0：" + Mathf.Round(-7.5f));
    }
}
```

在这段代码中，分别打印出了当小数部分b小于、大于和等于0.5时方法Round的返回值，其输出结果如图5-14所示，对输出结果的计算方法请参考功能说明。

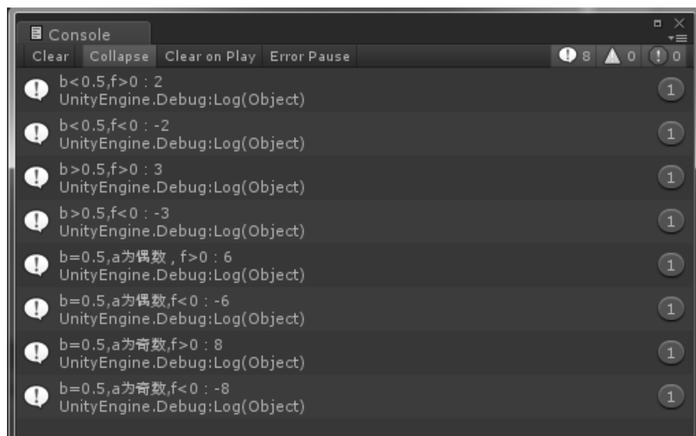


图5-14　Round实例演示的运行结果

5.2.12 SmoothDamp方法：模拟阻尼运动

基本语法 (1) public static float SmoothDamp(float current, float target, ref float currentVelocity, float smoothTime);

(2) public static float SmoothDamp(float current, float target, ref float currentVelocity, float smoothTime, float maxSpeed);

(3) public static float SmoothDamp(float current, float target, ref float currentVelocity, float smoothTime, float maxSpeed, float deltaTime);

对以上重载方法中涉及的参数进行说明：参数current为起始值；参数target为目标值；参数currentVelocity为当前帧速度，ref类型；参数smoothTime为预计平滑时间；参数maxSpeed为当前帧最大速度值，默认值为Mathf.Infinity；参数deltaTime为平滑时间，值越大返回值也相对越大，一般用Time.deltaTime计算。

5

功能说明 此方法的功能是模拟平滑阻尼运动，并返回模拟插值。smoothTime : float，预计平滑时间，物体越靠近目标，加速度的绝对值越小。实际到达目标的时间往往要比预计时间大很多，建议smoothTime的取值范围为(0.0f,1.0f)，若想控制物体到达目标的时间可以通过控制maxSpeed来达到目的。maxSpeed : float = Mathf.Infinity，每帧返回值的最大值，默认值为Mathf.Infinity。

提　示 可以观察实例演示中maxSpeed取默认值和取较小值时当前速度随时间变化的示意图。

实例演示 下面通过实例演示方法SmoothDamp的使用。

```
using UnityEngine;
using System.Collections;

public class SmoothDamp_ts : MonoBehaviour
{
    float targets = 110.0f;//目标值
    float cv1 = 0.0f, cv2 = 0.0f; //输出值
    float maxSpeeds = 50.0f;//每帧最大值
    float f1 = 10.0f, f2 = 10.0f;//起始值
    void FixedUpdate()
    {
        //maxSpeed取默认值
        f1 = Mathf.SmoothDamp(f1, targets, ref cv1, 0.5f);
        Debug.Log("f1:" + f1);
        Debug.Log("cv1:" + cv1);
        //maxSpeed取有限值50.0f
        f2 = Mathf.SmoothDamp(f2, targets, ref cv2, 0.5f, maxSpeeds);
        Debug.Log("f2:" + f2);
        Debug.Log("cv2:" + cv2);
    }
}
```

在这段代码的FixedUpdate方法中调用了两次SmoothDamp方法，第一次调用时取maxSpeed值为默认值，即无穷大，第二次调用时取maxSpeed值为有限值，然后分别打印出它们每帧的输出值。图5-15和图5-16分别是对输出值cv1和cv2随时间变化的可视化显示，可以发现，起始时输出速度提升很快，结束时速度下降却很平缓，在移动距离相同的情况下，有最大速度限制的花费时间也更多。

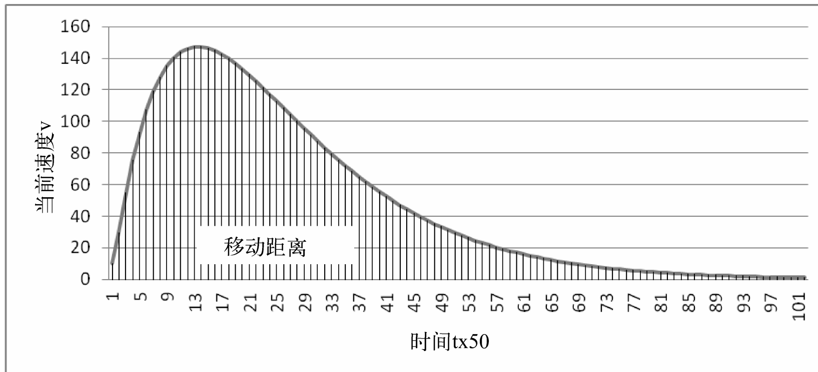


图5-15　当maxSpeed取默认值时速度cv1随时间变化的示意图

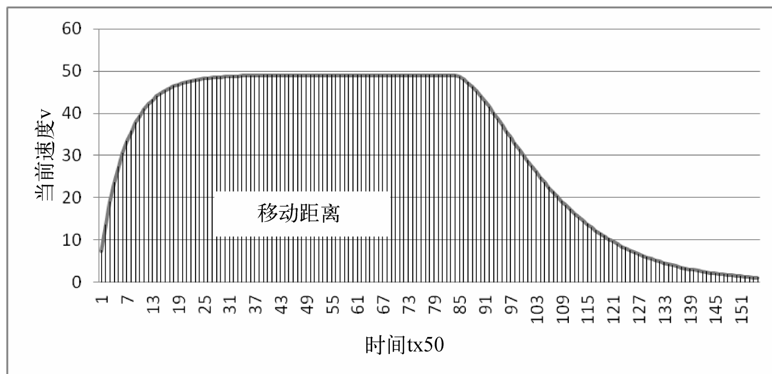


图5-16　当maxSpeed取较小值50.0f时速度cv2随时间变化的示意图

5.2.13　SmoothDampAngle方法：阻尼旋转

基本语法　(1) `public static float SmoothDampAngle(float current, float target, ref float current Velocity, float smoothTime);`

(2) `public static float SmoothDampAngle(float current, float target, ref float current Velocity, float smoothTime, float maxSpeed);`

(3) `public static float SmoothDampAngle(float current, float target, ref float current Velocity, float smoothTime, float maxSpeed, float deltaTime);`

对以上重载方法中涉及的参数进行说明：参数current为起始值；参数target为目标值；参数currentVelocity为当前帧速度，ref类型；参数smoothTime为预计平滑时间；参数maxSpeed为当前帧最大速度值，默认值为Mathf.Infinity；参数deltaTime为平滑时间，值越大返回值也相对越大，一般用Time.deltaTime计算。

功能说明 此方法的功能是模拟角度的平滑阻尼旋转，并返回模拟插值。此方法与SmoothDamp方法功能类似，但此方法主要用来模拟角度旋转的平滑阻尼运动。

实例演示 下面通过实例演示方法SmoothDampAngle的使用。

```
using UnityEngine;
using System.Collections;

public class SmoothDampAngle_ts : MonoBehaviour
{
    public Transform targets;
    float smoothTime = 0.3f;
    float distance = 5.0f;
    float yVelocity = 0.0f;
    void Update()
    {
        //返回平滑阻尼角度值
        float yAngle = Mathf.SmoothDampAngle(transform.eulerAngles.y, targets.eulerAngles.y,
            ref yVelocity, smoothTime);
        Vector3 positions = targets.position;
        //由于使用transform.LookAt，此处计算targets的-z轴方向距离targets为distance
        //欧拉角为摄像机绕target的y轴旋转yAngle的坐标位置
        positions += Quaternion.Euler(0, yAngle, 0) * new Vector3(0, 0, -distance);
        //向上偏移2个单位
        transform.position = positions + new Vector3(0.0f, 2.0f, 0.0f);
        transform.LookAt(targets);
    }
    void OnGUI()
    {
        if (GUI.Button(new Rect(10.0f, 10.0f, 200.0f, 45.0f), "将targets旋转60度"))
        {
            //更改targets的eulerAngles
            targets.eulerAngles += new Vector3(0.0f, 60.0f, 0.0f);
        }
    }
}
```

在这段代码的Update方法中，首先调用方法SmoothDampAngle计算出一个从摄像机当前角度到targets角度的插值，然后使用这个插值求解摄像机从targets的-z轴方向distance距离处偏移yAngle角度的位置，最后使用方法LookAt使摄像机朝向targets目标，请自行运行程序查看。

5.2.14 SmoothStep方法：平滑插值

基本语法 `public static float SmoothStep(float from, float to, float t);`

其中参数from为起始值，参数to为结束值，参数t为插值系数。

功能说明　此方法的功能是返回一个从from到to的平滑插值。对其使用方法说明如下。

- 参数from和to是两个任意的float类型数值，它们之间没有任何大小约束关系，from可以大于to也可以小于to。
- 插值系数t的有效范围为[0.0f,1.0f]，当t<0时其有效值t'为0.0f，当t>1时其有效值t'为1.0f。

实例演示　下面通过实例演示方法SmoothStep的使用。

```
using UnityEngine;
using System.Collections;

public class SmoothStep_ts : MonoBehaviour {
    float min = 10.0f;
    float max = 110.0f;
    float f1, f2=0.0f;

    void FixedUpdate () {
        //f1为SmoothStep插值返回值
        f1 = Mathf.SmoothStep(min,max,Time.time);
        //计算相邻两帧插值的变化
        f2 = f1 - f2;
        Debug.Log("f1:"+f1);
        Debug.Log("f2:" + f2);
        f2 = f1;
    }
}
```

在这段代码中，首先声明了两个变量f1和f2，用于记录当前帧和前一帧的插值记录，然后在FixedUpdate方法中调用方法SmoothStep，并将返回值赋给f1，接着计算与前一帧插值的差，并分别打印出f1和f2的值，最后再将f1赋给f2。图5-17和图5-18是对f1和f2输出值的图形化表示。图5-17是f1值的变化，从图5-17中可看到，f1的变化呈现两头平缓而中间变化较大的状态，从图5-18中可进一步发现f1在两头值的递增较小而中间值递增较大的状况。

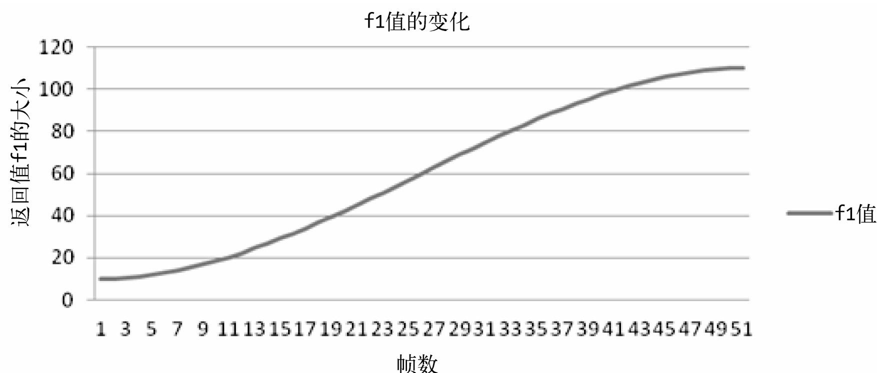


图5-17　输出结果f1的变化示意图

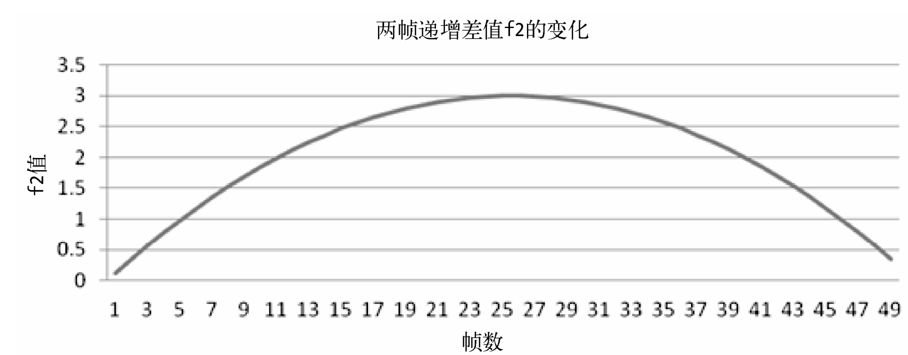


图5-18 输出结果f2的变化示意图

第 6 章

Matrix4x4类

6

在脚本中通常用Vector3、Quaternion、Transform等类的属性和方法来对物体进行变换，Matrix4x4类通常用在一些比较特殊的地方，如对摄像机的非标准投影变换等。本章主要介绍了Matrix4x4类的一些实例方法和静态方法。

6.1 Matrix4x4 类实例方法

在Matrix4x4类中，涉及的实例方法有MultiplyPoint方法、MultiplyPoint3x4方法、MultiplyVector方法和SetTRS方法，下面将详细介绍这些方法。

6.1.1 MultiplyPoint方法：投影矩阵变换

基本语法 public Vector3 MultiplyPoint(Vector3 v);

功能说明 此方法的功能是用来对点v进行投影矩阵变换。例如，设m1为Matrix4x4实例，v1为Vector3实例，Vector3 v2=m1. MultiplyPoint(v1)，则v2值的变换过程如下：v2=v1·m1·M，系统在进行变换时会给v1增加一个w的分量，扩充为四维向量，w默认值为1，而M为投影变换矩阵：

$$M=\begin{bmatrix} \frac{\cot\theta}{Aspect} & 0 & 0 & 0 \\ 0 & \cot\theta & 0 & 0 \\ 0 & 0 & \frac{f}{f-n} & 1 \\ 0 & 0 & \frac{n\cdot f}{n-f} & 0 \end{bmatrix},$$

其中f和n为远视口和近视口的距离，如图6-1所示，θ为视口夹角，Aspect为视口的纵横比值。

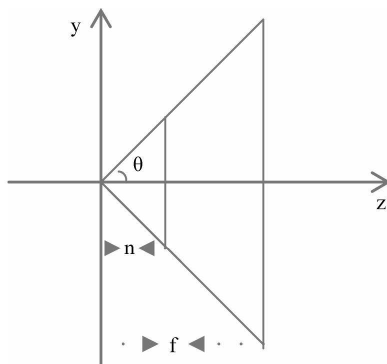


图6-1 Camera视口示意图

提 示 MultiplyPoint主要用于Camera的投影变换，对于一般物体的矩阵变换用MultiplyPoint3x4方法，不涉及投影变换，计算速度也更快。

实例演示 请参考Camera类中cameraToWorldMatrix属性的实例演示。

6.1.2 MultiplyPoint3x4 方法：矩阵变换

基本语法 `public Vector3 MultiplyPoint3x4(Vector3 v);`

功能说明 此方法的功能是用来对参数值点v进行矩阵变换。此方法在对参数v进行矩阵变换时，不涉及投影变换，速度比MultiplyPoint快。例如，设m1为Matrix4x4实例，v1为Vector3实例，则Vector3 v2=m1. MultiplyPoint3x4(v1)，即为v2=v3*m1，其中v3=(v1,w)，w默认值为1。

实例演示 请参考方法SetTRS的实例演示。

6.1.3 MultiplyVector方法：矩阵变换

基本语法 `public Vector3 MultiplyVector(Vector3 v);`

功能说明 此方法的功能是用来对方向向量v进行矩阵变换。此方法与方法MultiplyPoint功能类似，但它把v当做方向向量而非坐标点进行变换，当用矩阵与v进行变换时，只是对v的方向进行转换，也即系统会对参与变换的Matrix4x4进行特殊处理：设M为参与变换的Matrix4x4实例，其值为：

$$M=\begin{bmatrix} m00 & m01 & m02 & m03 \\ m10 & m11 & m12 & m13 \\ m20 & m21 & m22 & m23 \\ m30 & m31 & m32 & m33 \end{bmatrix},$$

则系统处理后的M值为：

$$M' = \begin{bmatrix} n00 & n01 & n02 & 0 \\ n10 & n11 & n12 & 0 \\ n20 & n21 & n22 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

其中 $n00^2+n10^2+n20^2=1$， $n01^2+n11^2+n21^2=1$， $n02^2+n12^2+n22^2=1$ 。此处以DirectX为例。在DirectX中向量变换是v・M，而在OpenGL中向量变换形式是M・v。

实例演示　下面通过实例演示方法MultiplyVector的使用。

```
using UnityEngine;
using System.Collections;

public class MultiplyVector_ts : MonoBehaviour
{
    public Transform tr;
    Matrix4x4 mv0 = Matrix4x4.identity;
    Matrix4x4 mv1 = Matrix4x4.identity;

    void Start()
    {
        //分别设置变换矩阵mv0和mv1的位置变换和角度变换都不为0
        mv0.SetTRS(Vector3.one * 10.0f, Quaternion.Euler(new Vector3(0.0f, 30.0f, 0.0f)),
            Vector3.one);
        mv1.SetTRS(Vector3.one * 10.0f, Quaternion.Euler(new Vector3(0.0f, 0.6f, 0.0f)),
            Vector3.one);
    }

    void Update()
    {
        //用tr来定位变换后的向量
        tr.position = mv1.MultiplyVector(tr.position);
    }
    void OnGUI()
    {
        if (GUI.Button(new Rect(10.0f, 10.0f, 120.0f, 45.0f), "方向旋转30度"))
        {
            Vector3 v = mv0.MultiplyVector(new Vector3(10.0f, 0.0f, 0.0f));
            //打印旋转后的向量，其方向发生了旋转
            Debug.Log("变换后向量："+v);
            //打印旋转后向量的长度
            //尽管mv0的位置变换不为0，但变换后向量的长度应与变换前相同
            Debug.Log("变换后向量模长：" + v.magnitude);
        }
    }
}
```

在这段代码中，首先实例化了两个Matrix4x4变量mv0和mv1，并在Start方法中对这两个变量进行重设，然后在Update方法中演示mv1的变换，在OnGUI方法中演示mv0的变

换，具体请参考代码注释。请自行运行程序查看，图6-2是Dubug输出截图。

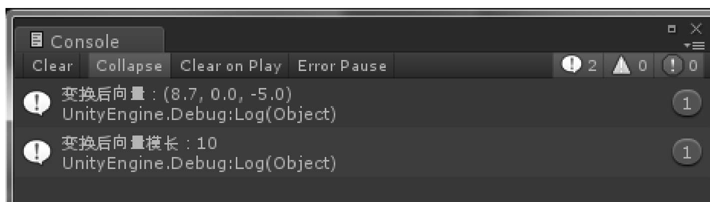


图6-2 方法MultiplyVector实例演示输出截图

6.1.4 SetTRS方法：重设Matrix4x4 变换矩阵

基本语法 public void SetTRS(Vector3 pos, Quaternion q, Vector3 s);

其中参数pos为位置向量，参数q为旋转角，参数s为放缩向量。

功能说明 此方法用来重设Matrix4x4变换矩阵。设有如下代码：

```
Matrix4x4 m1=Matrix4x4.identity;
m1. SetTRS (pos, q, s);
Vector3 v2=m1.MultiplyPoint3x4(v1);
```

则v2的值等同于将v1的position增加pos，rotation旋转q，scale放缩s后的值。

实例演示 下面通过实例演示方法SetTRS的使用。

```
using UnityEngine;
using System.Collections;

public class SetTRS_ts : MonoBehaviour
{
    Vector3 v1 = Vector3.one;
    Vector3 v2 = Vector3.zero;

    void Start()
    {
        Matrix4x4 m1 = Matrix4x4.identity;
        //Position沿y轴增加5个单位，绕y轴旋转45度，放缩2倍
        m1.SetTRS(Vector3.up * 5, Quaternion.Euler(Vector3.up * 45.0f), Vector3.one * 2.0f);
        //也可以使用如下静态方法设置m1变换
        //m1 = Matrix4x4.TRS(Vector3.up * 5, Quaternion.Euler(Vector3.up * 45.0f), Vector3.one
            * 2.0f);
        v2 = m1.MultiplyPoint3x4(v1);
        Debug.Log("v1的值：" + v1);
        Debug.Log("v2的值：" + v2);
    }

    void FixedUpdate()
    {
        Debug.DrawLine(Vector3.zero, v1, Color.green);
        Debug.DrawLine(Vector3.zero, v2, Color.red);
```

```
    }
}
```

在这段代码中，首先声明了两个变量v1和v2，其中v1的值为Vector3.one，然后在Start方法中初始化一个Matrix4x4变量m1，并调用方法SetTRS重置变量m1，接着调用方法MultiplyPoint3x4对向量v1进行变换，并将变换后的值赋给v2，最后在FixedUpdate方法中根据v1和v2的值绘制了两条直线，请自行运行程序查看。

代码中从v1向v2变换的过程如下:首先v1绕y轴旋转45度后变为Vector3（1.414,1.0,0.0），即x轴的分量变为了原来x和z分量的长度值，y值不变。接着对v1扩大2倍后v1变为Vector3（2.8,2.0,0.0）。最后将v1沿着y轴增加5个单位后变为Vector3（2.8,7.0,0.0），即为最后变换结果，图6-3是程序运行结果。

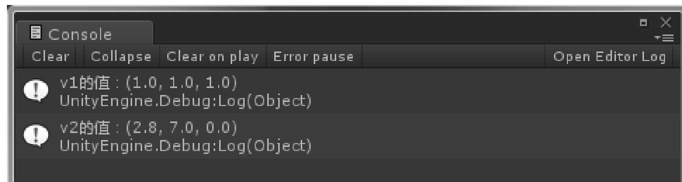


图6-3　实例演示程序运行结果

6.2　Matrix4x4 类静态方法

在Matrix4x4类中，涉及的静态方法有Ortho方法、Perspective方法和TRS方法，下面详细介绍这3个方法。

6.2.1　Ortho方法：创建正交投影矩阵

基本语法　public static Matrix4x4 Ortho(float left, float right, float bottom, float top, float zNear, float zFar);

其中参数left为正交视口左边边长，参数right为正交视口右边边长，参数bottom为正交视口下部边长，参数top为正交视口上部边长，参数zNear为近视口距离，参数zFar为远视口距离。

功能说明　此方法的功能是创建一个正交投影矩阵，其各个参数解释如图6-4和图6-5所示。

提　　示

- 参数left、right、bottom和top分正负方向，一般以right和top为正数，left和bottom为负数。
- left与right的值不能相等，bottom与top的值也不能相等，否则程序会报错。
- 为防止视图变形，参数的设定通常需要和Camera的aspect结合使用。

实例演示　请参考方法Perspective的实例演示。

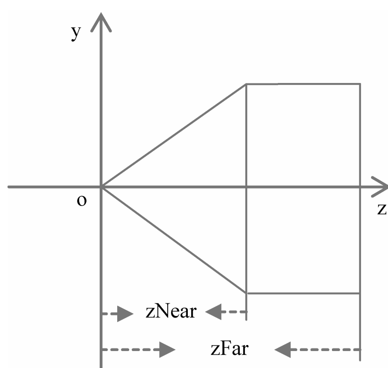


图6-4 正交投影视口示意图01

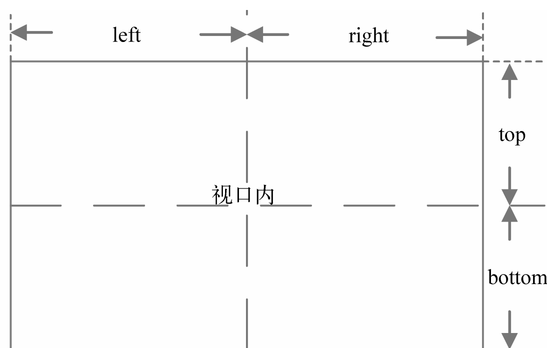


图6-5 正交投影视口示意图02

6.2.2 Perspective方法：创建透视投影矩阵

基本语法 `public static Matrix4x4 Perspective(float fov, float aspect, float zNear, float zFar);`

其中参数fov为视口夹角，参数aspect为视口纵横比例，参数zNear为近视口距离，参数zFar为远视口距离。

功能说明 此方法的功能是创建一个透视投影矩阵。若要更改摄像机的透视投影矩阵，可以用如下代码：

```
Camera.main. projectionMatrix=Matrix4x4.Perspective(fov,aspect,zNerar,zFar);
```

若要重置其投影矩阵，需要使用代码：

```
Camera.main. ResetProjectionMatrix ();
```

方法中各个参数图解如图6-6和图6-7所示（其中aspect=width/height）。

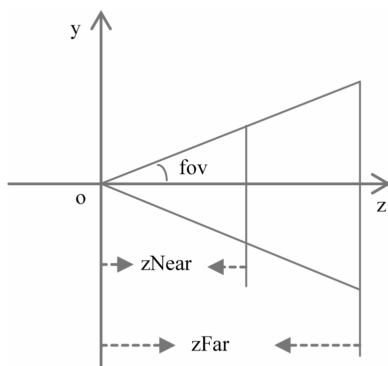


图6-6 透视投影视口示意图01



图6-7 透视投影视口示意图02

实例演示 下面通过实例演示方法Perspective的使用。

6

```
using UnityEngine;
using System.Collections;

public class OrthoAndPerspective_ts : MonoBehaviour
{
    Matrix4x4 Perspective = Matrix4x4.identity;//透视投影变量
    Matrix4x4 ortho = Matrix4x4.identity;//正交投影变量
    //声明变量，用于记录正交视口的左、右、下、上的值
    float l, r, b, t;
    void Start()
    {
        //设置透视投影矩阵
        Perspective = Matrix4x4.Perspective(65.0f, 1.5f, 0.1f, 500.0f);
        t = 10.0f;
        b = -t;
        //为防止视图变形需要与 Camera.main.aspect相乘
        l = b * Camera.main.aspect;
        r = t * Camera.main.aspect;
        //设置正交投影矩阵
        ortho = Matrix4x4.Ortho(l, r, b, t, 0.1f, 100.0f);
    }

    void OnGUI()
    {
        //使用默认正交投影
        if (GUI.Button(new Rect(10.0f, 8.0f, 150.0f, 20.0f), "Reset Ortho"))
        {
            Camera.main.orthographic = true;
            Camera.main.ResetProjectionMatrix();
            Camera.main.orthographicSize = 5.1f;
        }
        //使用自定义正交投影
        if (GUI.Button(new Rect(10.0f, 38.0f, 150.0f, 20.0f), "use Ortho"))
        {
            ortho = Matrix4x4.Ortho(l, r, b, t, 0.1f, 100.0f);
            Camera.main.orthographic = true;
            Camera.main.ResetProjectionMatrix();
            Camera.main.projectionMatrix = ortho;
            Camera.main.orthographicSize = 5.1f;
        }
        //使用自定义透视投影
        if (GUI.Button(new Rect(10.0f, 68.0f, 150.0f, 20.0f), "use Perspective"))
        {
            Camera.main.orthographic = false;
            Camera.main.projectionMatrix = Perspective;
        }
        //恢复系统默认透视投影
        if (GUI.Button(new Rect(10.0f, 98.0f, 150.0f, 20.0f), "Reset  Perspective"))
        {
            Camera.main.orthographic = false;
            Camera.main.ResetProjectionMatrix();
        }
    }
}
```

在这段代码中，首先初始化了两个Matrix4x4变量Perspective和ortho，然后在Start方法中创建了一个自定义的透视投影矩阵和一个正交投影矩阵，并将其赋给变量Perspective和ortho，最后在OnGUI方法中编写了4种不同的投影方式：正交投影、自定义正交投影、自定义透视投影和系统默认透视投影，请自行运行程序查看不同投影模式下的视图。

6.2.3 TRS方法：返回Matrix4x4 实例

基本语法 public static Matrix4x4 TRS(Vector3 pos, Quaternion q, Vector3 s);

其中参数pos为位置向量，参数q为旋转角，参数s为放缩向量。

功能说明 此方法使用pos、q和s作为变换参数返回一个Matrix4x4实例。设有如下代码：

```
Matrix4x4 m1=Matrix4x4. TRS (pos, q, s);
Vector v2=m1.MultiplyPoint3x4(v1);
```

则v2的值等同于将v1的position增加pos，rotation旋转q，scale放缩s后的值。

实例演示 请参考方法SetTRS的实例演示。

第 7 章

Object类

7

Object类是Unity中所有对象的基类，例如GameObject、Component、Material、Shader、Texture、Mesh、Font等都是Object的子类。本章主要介绍了Object类的一些实例方法和静态方法。

7.1 Object 类实例方法

在Object类中，涉及的实例方法主要有GetInstanceID 方法，下面详细介绍这个方法。

GetInstanceID方法：Object对象ID

基本语法 public int GetInstanceID();

功能说明 此方法用来返回Object对象的实例化ID。对其使用功能说明如下。

- 每个Object对象的实例、Object子类的实例如GameObject、Component等以及Object子类的子类的实例如Transform、Rigidbody等在工程中都有唯一的ID（int类型）。并且从程序开始运行到结束，除非对象被销毁，否则每个实例对应的ID都不会改变。
- 从GameObject.CreatePrimitive()或Object.Instantiate()中创建或克隆的每个名字相同的GameObject对象都有唯一的ID，即虽然名字相同，但ID却是不同的。在游戏开发中有时需要克隆大量的物体，而每个物体的生命周期需要单独记录，此时这两种方法很有用。

实例演示 下面通过实例演示方法GetInstanceID的使用。

```
using UnityEngine;
using System.Collections;

public class GetInstanceID_ts : MonoBehaviour {
  void Start () {
        Debug.Log("gameObject的ID："+gameObject.GetInstanceID());
        Debug.Log("transform的ID："+transform.GetInstanceID());

        GameObject g1, g2;
        //从GameObject创建一个对象
        g1=GameObject.CreatePrimitive(PrimitiveType.Cube);
```

```
            //克隆对象
            g2=Instantiate(g1,Vector3.zero,Quaternion.identity)as GameObject;

            Debug.Log("GameObject g1的ID："+g1.GetInstanceID());
            Debug.Log("Transform g1的ID："+g1.transform.GetInstanceID());

            Debug.Log("GameObject g2的ID：" + g2.GetInstanceID());
            Debug.Log("Transform g2的ID：" + g2.transform.GetInstanceID());
        }
    }
```

在这段代码中，首先分别打印出了gameObject和transform的InstanceID，然后分别创建和实例化了两个新对象g1和g2，最后分别打印出了g1和g2的对象ID和组件ID。图7-1为程序输出结果，从输出结果可以发现，gameObject的InstanceID和gameObject下的组件如Transform的InstanceID是不相同的。

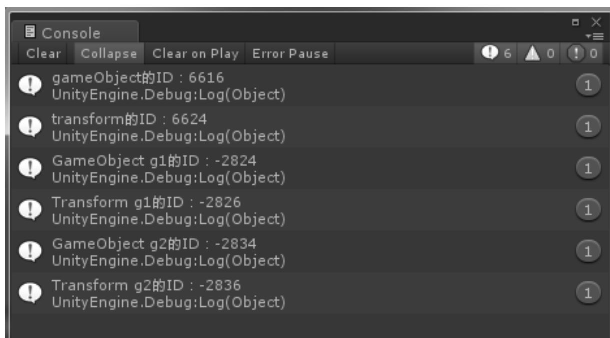


图7-1　GetInstanceID实例演示运行结果

7.2　Object 类静态方法

在Object类中，涉及的静态方法有Destroy方法、DontDestroyOnLoad方法、FindObjectOfType方法、FindObjectsOfType方法和Instantiate方法，由于FindObjectOfType方法和FindObjectsOfType方法功能相似，因此放到一起介绍，下面详细介绍这些方法。

7.2.1　Destroy方法：销毁对象

基本语法　(1) public static void Destroy(Object obj);

(2) public static void Destroy(Object obj, float t);

其中参数obj为待销毁的对象，参数t为销毁延迟时间，默认为0。

功能说明　此方法的功能是在执行完本方法t秒后销毁obj对象。方法Destroy可以销毁一个GameObject对象，也可以销毁GameObject对象中的某个组件如Rigidbody、脚本等，但是除非销毁整个GameObject对象，否则不可单独销毁Transform组件。当销毁整个

GameObject时GameObject的所有组件及子类将一并被销毁。

提　示　与此方法功能相近的方法有DestroyImmediate和DestroyObject。其中方法DestroyImmediate可以立即销毁某个Object对象及其在Assets中的资源文件，编程中慎用，推荐使用Destroy方法代替。

实例演示　下面通过实例演示方法Destroy的使用。

```
using UnityEngine;
using System.Collections;

public class Destroy_ts : MonoBehaviour {
    public GameObject GO,Cube;
    void Start () {
        //5秒后销毁GO对象的Rigidbody组件
        Destroy(GO.rigidbody,5.0f);
        //7秒后销毁GO对象中的Destroy_ts脚本
        Destroy(GO.GetComponent<Destroy_ts>(),7.0f);
        //10秒后销毁Cube对象，同时Cube对象的所有组件及子类将一并销毁
        Destroy(Cube, 10.0f);
    }
}
```

在这段代码中，首先声明了两个变量GO和Cube，然后在Start方法中依次销毁了GO对象的Rigidbody组件、GO对象的脚本组件和Cube对象。运行程序可以发现，销毁对象的某个组件时，对象自身不会被销毁，但销毁某个对象时，其所包含的组件将一并被销毁。

7.2.2　DontDestroyOnLoad 方法：新场景中保留对象

基本语法　public static void DontDestroyOnLoad(Object target);

其中参数target为被保留的对象。

功能说明　此方法用于设置参数target指向的对象是否在新Scene中被保留下来。对其使用说明如下。

- 如果target为根物体的GameObject对象或GameObject对象中的某个组件，则物体自身及其子物体都会被导入到新Scene中，当然它们也可以在新Scene中进行编辑操作。
- 如果target不为根物体的GameObject对象或GameObject对象中的某个组件，则此方法将失效，即target及其子物体不会被导入到新Scene中。若想把场景中某个子物体导入到新Scene中，可以用Transform.DetachChildren方法进行父子层级关系分离，然后再导入新Scene中。

实例演示　下面通过实例演示方法DontDestroyOnLoad的使用。

```
using UnityEngine;
using System.Collections;

public class DontDestoryOnLoad_ts : MonoBehaviour
```

```
{
    public GameObject g1, g2;
    public Renderer re1, re2;
    void Start()
    {
        //g1指向一个顶层父物体对象,在导入新Scene时g1被保存
        DontDestroyOnLoad(g1);
        //g2指向一个子类对象,在导入新Scene时会发现g2没有被保存
        DontDestroyOnLoad(g2);
        //re1指向一个顶层父物体的Renderer组件,在导入新Scene时re1被保存
        DontDestroyOnLoad(re1);
        //re2指向一个子类对象的renderer组件, 在导入新Scene时会发现re2指向的对象及组件没有被
            保存
        DontDestroyOnLoad(re2);
        Application.LoadLevel("FindObjectsOfType_unity");
    }
}
```

在这段代码中，首先声明了4个不同类型的变量g1、g2、re1和re2，它们所指向的对象如图7-2所示，然后在Start方法中分别对这4个对象执行方法DontDestroyOnLoad，最后导入新场景。场景内容如图7-3所示，只有G1、Re1和它们的子类被保留到了新的场景中。

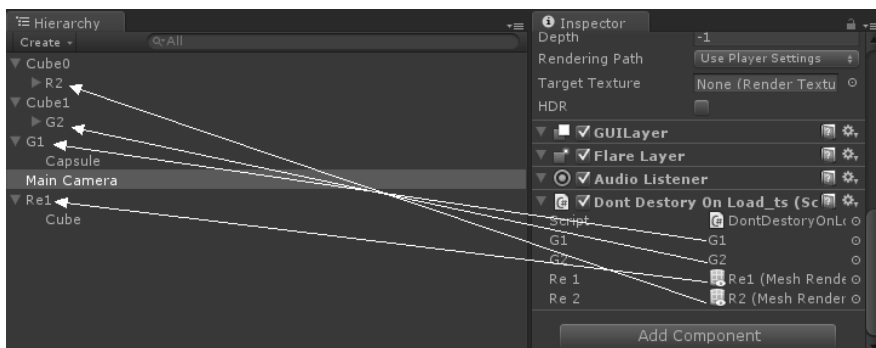


图7-2　开始的Scene

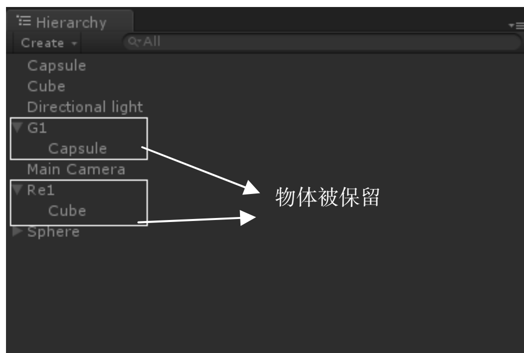


图7-3　新导入的Scene

7

7.2.3 FindObjectsOfType方法：获取对象

基本语法 (1) public static T[] FindObjectsOfType<T>() where T : Object;

(2) public static Object[] FindObjectsOfType(Type type);

其中参数type为要获取的对象类型，可以是GameObject类型或Component类型。

功能说明 此方法用于获取工程中所有符合参数类型的对象。此方法需要遍历整个工程，执行速度较慢，不适宜在每帧中调用。对于遍历的结果可以通过对象的name或InstanceID等属性进行有选择地处理。

提　　示 FindObjectOfType方法与此方法功能相近，用于获取工程中符合type类型的第一个对象，多用于检测工程中是否含有某种类型的对象。

实例演示 下面通过实例演示方法FindObjectsOfType的使用。

```
using UnityEngine;
using System.Collections;

public class FindObjectOfType_ts : MonoBehaviour {
    void Start () {
        GameObject[] gos = FindObjectsOfType(typeof(GameObject)) as GameObject[];
        foreach(GameObject go in gos){
            //1.5秒后销毁除摄像机外的所有GameObject
            if (go.name != "Main Camera")
            {
                Destroy(go, 1.5f);
            }
        }

        Rigidbody[] rbs = FindObjectsOfType(typeof(Rigidbody))as Rigidbody[];
        foreach(Rigidbody rb in rbs){
            //启用除球体外的所有刚体的重力感应
            if(rb.name!="Sphere"){
                rb.useGravity = true;
            }
        }
  }
}
```

在这段代码中，首先调用方法FindObjectsOfType查找游戏中所有的GameObject对象，并将查找结果赋给数组gos，然后遍历数组gos，在1.5秒后销毁除摄像机外的所有GameObject对象。然后再次调用方法FindObjectsOfType查找游戏中所有的Rigidbody对象，并将查找结果赋给数组rbs，然后遍历数组rbs，启用除Sphere外所有刚体的重力感应。请自行运行程序查看。

7.2.4 Instantiate方法：实例化对象

基本语法 (1) public static Object Instantiate(Object original);

(2) public static Object Instantiate(Object original, Vector3 position, Quaternion rotation);

其中参数original为实例化对象的类型，参数position为实例化对象的位置，参数rotation为实例化对象的旋转角度。

功能说明 此方法用于实例化一个Object对象。Instantiate可以实例化Object、Object的子类以及Object子类的子类等。当实例化一个对象时，会同时实例化根对象的所有子类。

实例演示 下面通过实例演示方法Instantiate的使用。

```
using UnityEngine;
using System.Collections;

public class Instantiate_ts : MonoBehaviour {
    public GameObject A;
    public Transform B;
    public Rigidbody C;
    void Start () {
        Object g1 = Instantiate(A,Vector3.zero,Quaternion.identity) as Object;
        Debug.Log("克隆一个Object对象g1:"+g1);
        GameObject g2 = Instantiate(A, Vector3.zero, Quaternion.identity) as GameObject;
        Debug.Log("克隆一个GameObject对象g2:" + g2);
        Transform t1 = Instantiate(B, Vector3.zero, Quaternion.identity) as Transform;
        Debug.Log("克隆一个Transform对象t1:" + t1);
        Rigidbody r1 = Instantiate(C, Vector3.zero, Quaternion.identity) as Rigidbody;
        Debug.Log("克隆一个Rigidbody对象r1:" + r1);
    }
}
```

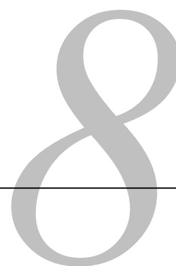
在这段代码中，首先声明了3个不同类型的公共变量A、B和C，然后在Start方法中调用Instantiate分别实例化了4种不同类型的对象，并将实例化的对象打印出来，程序运行结果如图7-4所示。



图7-4 Instantiate实例演示运行结果

第 8 章

Quaternion类



Quaternion又称四元数，由x、y、z和w这4个分量组成，属于struct类型。在Unity中，用Quaternion来存储和表示对象的旋转角度。Quaternion的变换比较复杂，对于GameObject一般的旋转及移动，可以用Transform中的相关方法实现。本章主要介绍了Quaternion类的一些实例属性、静态方法和运算符，并对Quaternion类相乘运算符“*”的两种重载格式在功能上的异同进行了简要的注解。

8.1 Quaternion 类实例属性

在Quaternion类中，涉及的实例属性主要有eulerAngles，下面详细介绍这个实例属性。

eulerAngles属性：欧拉角

基本语法 public Vector3 eulerAngles { get; set; }

功能说明 此属性用来返回或设置Quaternion实例对应的欧拉角，对其使用说明如下。

- 对GameObject对象的Transform进行欧拉角的变换次序是，先绕z轴旋转相应的角度，再绕x轴旋转相应的角度，最后再绕y轴旋转相应的角度。注意不同的旋转次序得到的最终状态是不同的。
- 对GameObject对象的旋转角进行赋值的方式通常有两种：第一种是将Quaternion实例赋值给transform的rotation，第二种是将三维向量代表的欧拉角直接赋值给transform的eulerAngles，见实例演示。

实例演示 下面通过实例演示属性eulerAngles的使用。

```
using UnityEngine;
using System.Collections;

public class EulerAngle_ts : MonoBehaviour
{
    public Transform A, B;
    Quaternion rotations=Quaternion.identity;
    Vector3 eulerAngle = Vector3.zero;
    float speed = 10.0f;
```

```
    void Update()
    {
        //第一种方式：将Quaternion赋值给transform的rotation
        rotations.eulerAngles = new Vector3(0.0f, speed * Time.time, 0.0f);
        A.rotation = rotations;
        //第二种方式：将三维向量代表的欧拉角直接赋值给transform的eulerAngles
        eulerAngle = new Vector3(0.0f, speed * Time.time, 0.0f);
        B.eulerAngles = eulerAngle;
    }
}
```

在这段代码中，我们演示了两种改变transform角度的方式，请自行运行程序查看。

8.2 Quaternion 类实例方法

在Quaternion类中涉及的实例方法有SetFromToRotation方法、SetLookRotation方法和ToAngleAxis方法，由于静态方法AngleAxis和实例方法ToAngleAxis功能相近，于是将这两个方法放到一起介绍。下面详细介绍这些方法。

8.2.1 SetFromToRotation方法：创建rotation实例

基本语法 public void SetFromToRotation(Vector3 fromDirection, Vector3 toDirection);

功能说明 此方法用于创建一个从fromDirection到toDirection的rotation。例如，设有以下代码：

```
Quaternion q1 = Quaternion.identity;
q1.SetFromToRotation(v1,v2);
transform.rotation = q1;
```

则相当于将GameObject对象进行如下变换：首先将GameObject对象自身坐标系的x、y、z轴方向和世界坐标系的x、y、z轴方向一致，然后将GameObject对象自身坐标系中向量v1指向的方向旋转到v2方向。

提　　示 不可直接使用transform.rotation.SetFromToRotation(v1,v2)方式进行设置，只能将实例化的Quaternion赋值给transform.rotation。

实例演示 下面通过实例演示方法SetFromToRotation的使用。

```
using UnityEngine;
using System.Collections;

public class SetFromToRotation_ts : MonoBehaviour {
    public Transform A, B, C;
    Quaternion q1 = Quaternion.identity;

  void Update () {
        //不可直接使用C.rotation.SetFromToRotation(A.position,B.position);
        q1.SetFromToRotation(A.position,B.position);
```

```
        C.rotation = q1;
        //在Scene面板中绘制直线
        Debug.DrawLine(Vector3.zero,A.position,Color.red);
        Debug.DrawLine(Vector3.zero, B.position, Color.green);
        Debug.DrawLine(C.position, C.position+new Vector3(0.0f,1.0f,0.0f), Color.black);
        Debug.DrawLine(C.position, C.TransformPoint(Vector3.up*1.5f), Color.yellow);
    }
}
```

在这段代码中，首先声明了3个GameObject类型的变量A、B和C，然后在Update方法中对q1调用方法SetFromToRotation，并将改变后的q1赋给C.rotation。注意不可直接使用C.rotation.SetFromToRotation(A.position,B.position)来设置C的rotation。最后在Scene场景中绘制直线以便更好地观察。请自行运行程序在Scene视图中查看，在程序运行时可以试着更改A或B的位置查看物体C的状态变化。图8-1是一张运行时截图及注释。

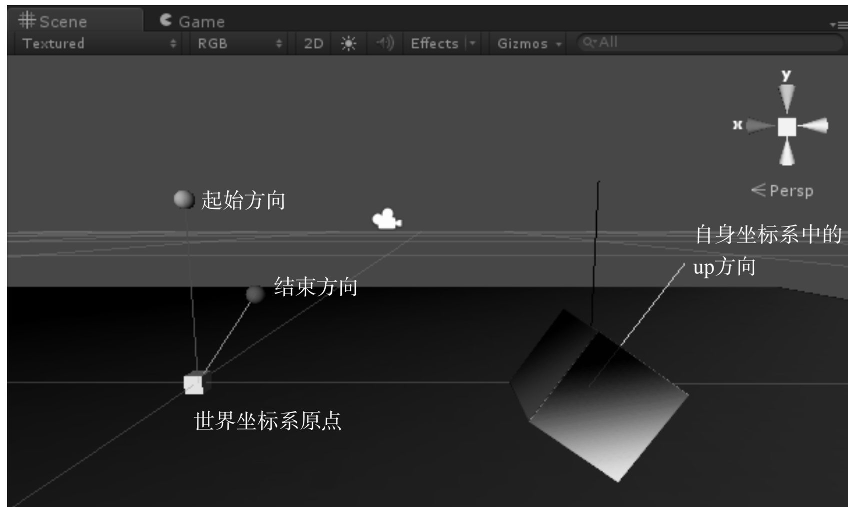


图8-1　SetFromToRotation实例演示运行时截图及注释

8.2.2　SetLookRotation方法：设置Quaternion实例的朝向

基本语法　(1) public void SetLookRotation(Vector3 view);

(2) public void SetLookRotation(Vector3 view, Vector3 up);

功能说明　此方法的功能是对一个Quaternion实例的朝向进行设置。设有如下代码：

```
Quaternion q1 = Quaternion.identity;
q1.SetLookRotation(v1, v2);
transform.rotation = q1;
```

则

- transform.forward方向与v1方向相同。
- transform.right垂直于由Vector3.zero、v1和v2这 3 点构成的平面。
- v2除了与Vector3.zero和v1构成平面来决定transform.right的方向外，还用来决定transform.up的朝向，因为当transform.forward和transform.right方向确定后，transform.up方向剩下两种可能，到底选用哪一种便由v2来影响。transform.up方向的选取方式总会使得transform.up的方向和v2方向的夹角小于或等于90度。当然，一般情况下v2.normalized和transform.up是不相同的。
- 当v1为Vector3.zero时，方法失效，即不要在使用此方法时把v1设置成Vector3.zero。

提　示　不可以直接使用transform.rotation.SetLookRotation(v1, v2)的方式来使用SetLookRotation方法，否则会不起作用。应该使用上述代码所示的方式，首先实例化一个Quaternion，然后对其使用SetLookRotation，最后将其赋给transform.rotation。

实例演示　下面通过实例演示方法SetLookRotation的使用。

```
using UnityEngine;
using System.Collections;

public class SetLookRotation_ts : MonoBehaviour
{
    public Transform A, B, C;
    Quaternion q1 = Quaternion.identity;

    void Update()
    {
        //不可直接使用C.rotation.SetLookRotation(A.position,B.position);
        q1.SetLookRotation(A.position, B.position);
        C.rotation = q1;
        //分别绘制A、B和C.right的朝向线
        //请在Scene视图中查看
        Debug.DrawLine(Vector3.zero, A.position, Color.red);
        Debug.DrawLine(Vector3.zero, B.position, Color.green);
        Debug.DrawLine(C.position, C.TransformPoint(Vector3.right * 2.5f), Color.yellow);
        Debug.DrawLine(C.position, C.TransformPoint(Vector3.forward * 2.5f), Color.gray);
        //分别打印C.right与A、B的夹角
        Debug.Log("C.right与A的夹角:" + Vector3.Angle(C.right, A.position));
        Debug.Log("C.right与B的夹角:" + Vector3.Angle(C.right, B.position));
        //C.up与B的夹角
        Debug.Log("C.up与B的夹角:" + Vector3.Angle(C.up, B.position));
    }
}
```

在这段代码中，首先声明了 3 个GameObject类型的变量A、B和C，然后在Update方法中对q1调用方法SetLookRotation，并将改变后的q1赋给C.rotation。注意不可直接使用C.rotation.SetLookRotation (A.position,B.position)来设置C的rotation。最后在Scene场景中绘制直线以便更好地观察。请自行运行程序在Scene视图中查看，在程序

运行时可以试着更改A或B的位置查看C的状态变化。图8-2是一张运行时截图及注释，图8-3是Debug的输出结果。

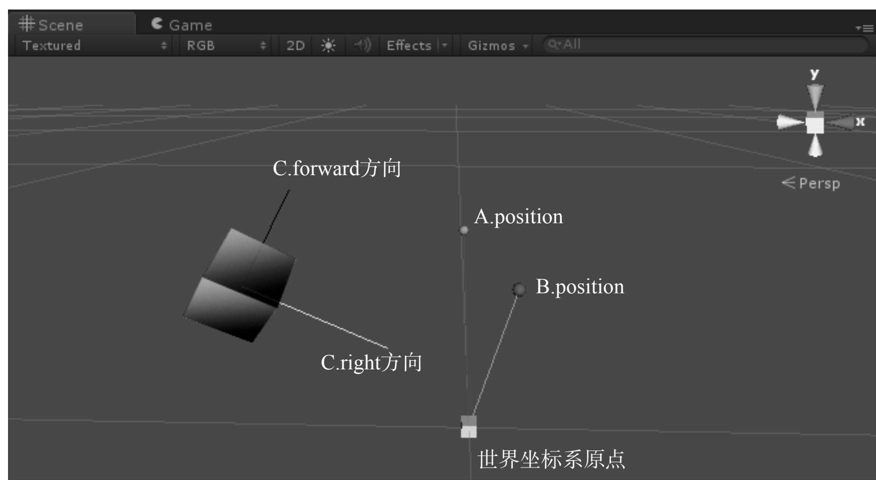


图8-2　SetLookRotation运行时截图及注释

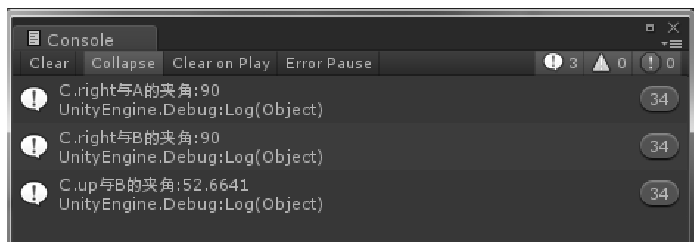


图8-3　SetLookRotation实例演示的运行结果

8.2.3　ToAngleAxis方法：Quaternion实例的角轴表示

基本语法　`public void ToAngleAxis(out float angle, out Vector3 axis);`

其中参数angle为旋转角，参数axis为轴向量。

功能说明　此方法用于将Quaternion实例转换为角轴表示。在transform.rotation.ToAngle Axis (out angle, out axis)中，输出值angle和axis的含义为：要将GameObject对象的rotation从Quaternion.Identity状态变换到当前状态，只需要将GameObject对象绕着axis的轴向（指世界坐标系中）旋转angle角度即可。

提　　示　此方法通常和静态方法AngleAxis (angle : float, axis : Vector3)联合使用，使得一个物体的rotation始终和另一个物体的rotation保持一致。

实例演示 下面通过实例演示方法ToAngleAxis的使用。

```
using UnityEngine;
using System.Collections;

public class ToAngleAxis_ts : MonoBehaviour {
    public Transform A, B;
    float angle;
    Vector3 axis = Vector3.zero;

void Update () {
    //使用ToAngleAxis获取A的Rotation的旋转轴和角度
    A.rotation.ToAngleAxis(out angle, out axis);
    //使用AngleAxis设置B的rotation，使得B的rotation状态的和A相同
    //当然也可以只使得B与A的axis相同，而angle不同
    //可以在程序运行时修改A的rotation查看B的状态
    B.rotation = Quaternion.AngleAxis(angle,axis);
}
}
```

在这段代码中，首先声明了两个GameObject类型的变量A和B，用于指向场景中的物体，然后在Update方法中，首先调用ToAngleAxis方法将A的rotation转换为角轴angle和axis，接着调用方法AngleAxis将角轴代表的Quaternion值赋给B.rotation。程序运行时可以在Inspector面板或Scene视图中随便修改A的旋转角，然后查看物体B的相应旋转状态，请自行运行程序查看。

8.3 Quaternion 类静态方法

在Quaternion类中涉及的静态方法有Angle方法、Dot方法、Euler方法、FromToRotation方法、Inverse方法、Lerp方法、LookRotation方法、RotateTowards方法和Slerp方法，下面详细介绍这些方法。

8.3.1 Angle方法：Quaternion实例间夹角

基本语法 public static float Angle(Quaternion a, Quaternion b);

功能说明 此方法用于返回从参数a到参数b变换的夹角。需要注意的是，返回的夹角不是某个局部坐标轴向变换的夹角，而是GameObject对象从状态a转换到状态b时需要旋转的最小夹角。

实例演示 下面通过实例演示方法Angle的使用。

```
using UnityEngine;
using System.Collections;

public class Angle_ts : MonoBehaviour {
    void Start()
    {
```

```
        Quaternion q1 = Quaternion.identity;
        Quaternion q2 = Quaternion.identity;
        q1.eulerAngles = new Vector3(10.0f, 20.0f, 30.0f);
        float f1 = Quaternion.Angle(q1,q2);
        float f2 = 0.0f;
        Vector3 v1 = Vector3.zero;
        q1.ToAngleAxis(out f2, out v1);

        Debug.Log("f1:" + f1);
        Debug.Log("f2:" + f2);
        Debug.Log("q1的欧拉角：" + q1.eulerAngles + " q1的rotation：" + q1);
        Debug.Log("q2的欧拉角：" + q2.eulerAngles + " q2的rotation：" + q2);
    }
}
```

在这段代码的Start方法中，首先实例化了两个Quaternion变量q1和q2，然后对q1的欧拉角进行赋值，接着调用方法Angle求q1和q2之间的夹角，并将返回值赋给f1，最后调用方法ToAngleAxis，求解从当前q1状态转换到Quaternion.identity状态需要旋转的最小角度值f2。图8-4是Debug输出结果，从输出结果可以发现f1和f2的值相等，即方法Angle的返回值是两个Quaternion参数间转换的最小夹角。

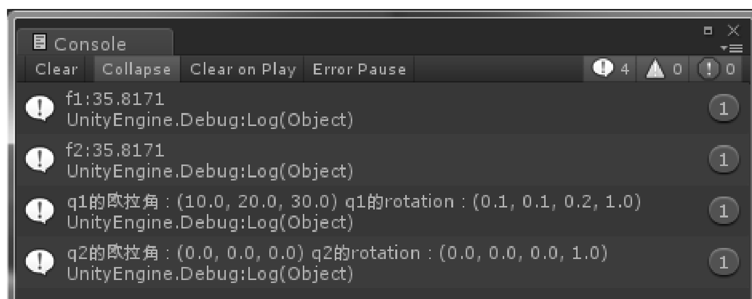


图8-4　Angle实例演示的运行结果

8.3.2　Dot方法：点乘

基本语法　public static float Dot(Quaternion a, Quaternion b);

功能说明　此方法用来求参数a和b的点乘。例如，设q1(x1,y1,z1,w1)、q2(x2,y2,z2,w2)为Quaternion的两个实例，则：

float f= Quaternion.Dot(q1,q2);

等价于f=x1*x2+y1*y2+z1*z2+w1*w2，结果值f的范围为[-1,1]。

- 当f=±1时，q1和q2对应的欧拉角是相等的，即在Game视图中看来它们的旋转状态是一样的；
- 当f=1时，它们的rotation相等；

- 当f=–1时，说明其中一个rotation比另一个多旋转了一周即360度。

实例演示 下面通过实例演示方法Dot的使用。

```
using UnityEngine;
using System.Collections;

public class Dot_ts : MonoBehaviour {
    public Transform A, B;
    Quaternion q1=Quaternion.identity;
    Quaternion q2=Quaternion.identity;
    float f;

    void Start () {
        A.eulerAngles = new Vector3(0.0f,40.0f,0.0f);
        //B比A绕y轴多转360度
        B.eulerAngles = new Vector3(0.0f, 360.0f+40.0f, 0.0f);
        q1 = A.rotation;
        q2 = B.rotation;
        f = Quaternion.Dot(q1,q2);
        Debug.Log("q1的rotation:"+q1);
        Debug.Log("q2的rotation:" + q2);
        Debug.Log("q1的欧拉角:" + q1.eulerAngles);
        Debug.Log("q2的欧拉角:" + q2.eulerAngles);
        Debug.Log("Dot(q1,q2):"+f);
    }
}
```

在这段代码中，首先声明了两个Transform变量，实例化了两个Quaternion变量，然后在Start方法中分别对A和B的eulerAngles属性进行赋值，并且B沿y轴的欧拉角要比A多360度，接着将A和B的rotation分别赋予q1和q2，最后调用Dot方法将q1和q2的点乘值赋给f。图8-5是Debug的输出截图，从输出结果可以发现，q1和q2的欧拉角相等，但它们的值却相反。当Dot返回值为–1时，两个参数的角度差值相差一周。

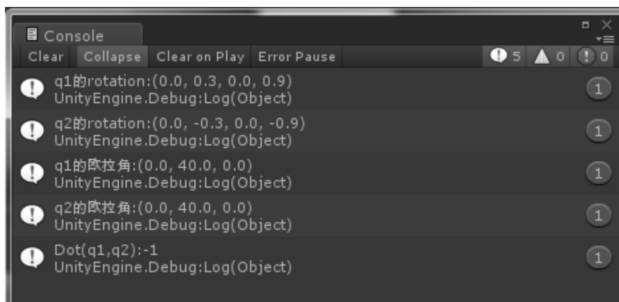


图8-5 Dot实例演示的运行结果

8.3.3 Euler方法：欧拉角对应的四元数

基本语法 (1) `public static Quaternion Euler(Vector3 euler);`

(2) public static Quaternion Euler(float x, float y, float z);

功能说明　此方法用于返回欧拉角Vector3(x,y,z)对应的四元数Quaternion实例。四元数Quaternion(qx,qy,qz,qw)与其欧拉角eulerAngles(ex,ey,ez)之间的对应关系如下。

已知PIover180=3.141592/180=0.0174532925f是计算机图形学中的一个常量，则变换过程如下。

ex=ex* PIover180/2.0f;
ey=ey* PIover180/2.0f;
ez=ez* PIover180/2.0f;

则：

qx=sin(ex)cos(ey)cos(ez)+cos(ex)sin(ey)sin(ez);
qy=cos(ex)sin(ey)cos(ez)- sin (ex)cos(ey)sin(ez);
qz=cos(ex)cos(ey)sin(ez)-sin (ex)sin(ey)cos(ez);
qw=cos(ex)cos(ey)cos(ez)+sin (ex)sin(ey)sin(ez);

实例演示　下面通过实例验证上述算法。

```
using UnityEngine;
using System.Collections;

public class Euler_ts : MonoBehaviour
{
    //记录欧拉角，单位为角度，可以在Inspector面板中设置
    public float ex, ey, ez;
    //用于记录计算结果
    float qx, qy, qz, qw;
    float PIover180 = 0.0174532925f;//常量
    Quaternion Q = Quaternion.identity;
    void OnGUI()
    {
        if (GUI.Button(new Rect(10.0f, 10.0f, 100.0f, 45.0f), "计算"))
        {
            Debug.Log("欧拉角：" + " ex：" + ex + "  ey：" + ey + "  ez：" + ez);
            //调用方法计算
            Q = Quaternion.Euler(ex, ey, ez);
            Debug.Log("Q.x:" + Q.x + " Q.y:" + Q.y + " Q.z:" + Q.z + " Q.w:" + Q.w);
            //测试算法
            ex = ex * PIover180 / 2.0f;
            ey = ey * PIover180 / 2.0f;
            ez = ez * PIover180 / 2.0f;
            qx = Mathf.Sin(ex) * Mathf.Cos(ey) * Mathf.Cos(ez) + Mathf.Cos(ex) *
                Mathf.Sin(ey) * Mathf.Sin(ez);
            qy = Mathf.Cos(ex) * Mathf.Sin(ey) * Mathf.Cos(ez) - Mathf.Sin(ex) *
                Mathf.Cos(ey) * Mathf.Sin(ez);
            qz = Mathf.Cos(ex) * Mathf.Cos(ey) * Mathf.Sin(ez) - Mathf.Sin(ex) *
                Mathf.Sin(ey) * Mathf.Cos(ez);
            qw = Mathf.Cos(ex) * Mathf.Cos(ey) * Mathf.Cos(ez) + Mathf.Sin(ex) *
```

```
                Mathf.Sin(ey) * Mathf.Sin(ez);
            Debug.Log(" qx:" + qx + " qy:" + qy + " qz:" + qz + " qw:" + qw);
        }
    }
}
```

在这段代码中，首先声明了 3 个公共变量ex, ey, ez，用于记录欧拉角，单位为角度，其值可以在Inspector面板中设置。然后在OnGUI方法中定义了一个Button，先调用Quaternion的静态方法Euler，打印出计算结果，然后再测试功能说明中的算法，并打印出计算结果。图8-6是一组数据的测试结果。

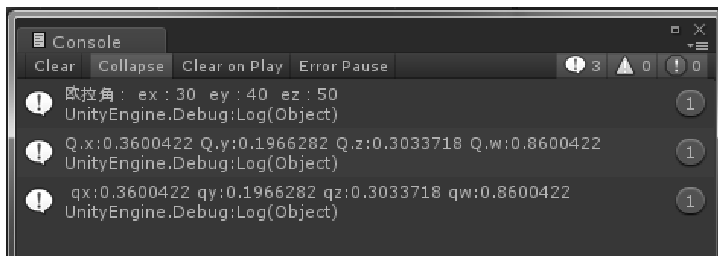


图8-6 方法Euler实例演示的运行截图

8.3.4 FromToRotation方法：Quaternion变换

基本语法 public static Quaternion FromToRotation(Vector3 fromDirection, Vector3 toDirection);

功能说明 此方法用来创建一个从参数fromDirection到toDirection的Quaternion变换。其功能和实例方法SetFromToRotation (fromDirection : Vector3, toDirection : Vector3)相同，只是用法上有些不同，请参考实例演示。

实例演示 下面通过实例演示方法FromToRotation的使用。

```
using UnityEngine;
using System.Collections;

public class FromToRotation_ts : MonoBehaviour
{
    public Transform A, B, C, D;
    Quaternion q1 = Quaternion.identity;

    void Update()
    {
        //使用实例方法
        //不可直接使用C.rotation.SetFromToRotation(A.position,B.position);
        q1.SetFromToRotation(A.position, B.position);
        C.rotation = q1;
        //使用类方法
        D.rotation = Quaternion.FromToRotation(A.position, B.position);
        //在Scene视图中绘制直线
```

```
            Debug.DrawLine(Vector3.zero, A.position, Color.white);
            Debug.DrawLine(Vector3.zero, B.position, Color.white);
            Debug.DrawLine(C.position, C.position + new Vector3(0.0f, 1.0f, 0.0f),
                Color.white);
            Debug.DrawLine(C.position, C.TransformPoint(Vector3.up * 1.5f), Color.white);
            Debug.DrawLine(D.position, D.position + new Vector3(0.0f, 1.0f, 0.0f),
                Color.white);
            Debug.DrawLine(D.position, D.TransformPoint(Vector3.up * 1.5f), Color.white);
        }
    }
```

在这段代码中，首先声明了4个Transform变量，用于指向场景中不同的对象，然后在Update方法中调用方法SetFromToRotation，使得C的旋转角度与向量A和B之间的夹角相同，接着使用方法FromToRotation，使得D的旋转角度与向量A和B之间的夹角相同，如图8-7所示。请自行运行程序在Scene视图而非Game视图中查看。程序运行时可以在Scene视图中试着拖动A或B的位置观察C和D的变化。

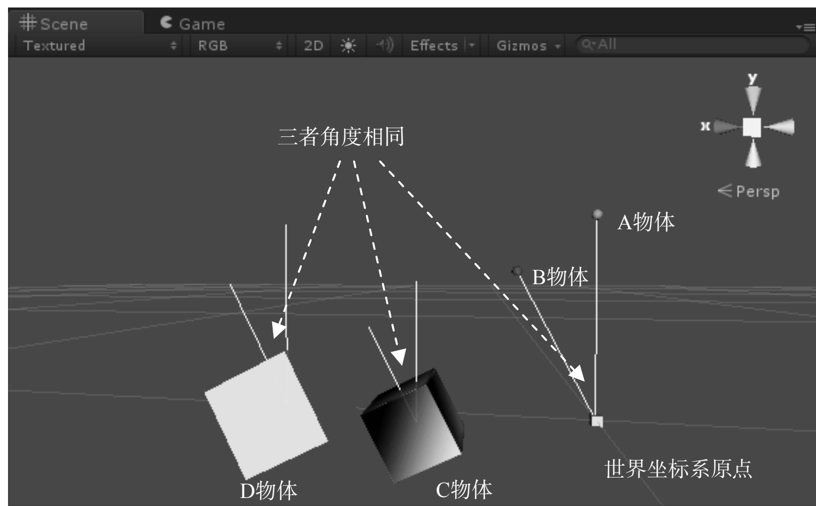


图8-7　实例演示中Scene面板截图

8.3.5　Inverse方法：逆向Quaternion值

基本语法　public static Quaternion Inverse(Quaternion rotation);

功能说明　此方法用于返回参数rotation的逆向Quaternion值。例如，设有实例rotation=(x,y,z,w)，则Inverse(rotation)=(−x, −y, −z,w)。从效果上说，设rotation.eulerAngles=(a,b,c)，则transform.rotation=Inverse(rotation)相当于transform依次绕自身坐标系的z轴、x轴和y轴分别旋转−c度、−a度和−b度。由于是局部坐标系内的变换，最后transform的欧拉角的各个分量值并不一定等于−a、−b或−c。

实例演示 下面通过实例演示方法Inverse的使用。

```
using UnityEngine;
using System.Collections;

public class Inverse_ts : MonoBehaviour
{
    public Transform A, B;
    void Start()
    {
        Quaternion q1 = Quaternion.identity;
        Quaternion q2 = Quaternion.identity;
        q1.eulerAngles = new Vector3(10.0f, 20.0f, 30.0f);
        q2 = Quaternion.Inverse(q1);

        A.rotation = q1;
        B.rotation = q2;

        Debug.Log("q1的欧拉角：" + q1.eulerAngles + " q1的rotation：" + q1);
        Debug.Log("q2的欧拉角：" + q2.eulerAngles + " q2的rotation：" + q2);
    }
}
```

在这段代码中，首先声明了两个变量A和B，用于指向场景中不同的物体对象，然后在Start方法中实例化了两个Quaternion实例q1和q2，接着调用方法Inverse求q1的逆向Quaternion值，并将返回值赋给q2，最后将q1和q2分别赋给A和B的rotation，并分别打印出q1和q2的欧拉角和rotation值。请自行运行程序查看，图8-8是一张Debug输出截图。

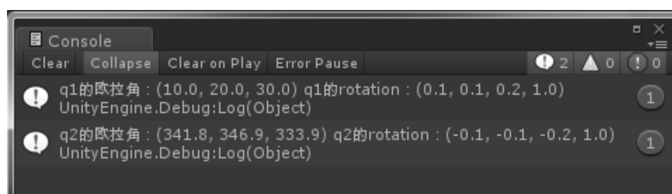


图8-8 Inverse实例演示的运行结果

8.3.6 Lerp方法：线性插值

基本语法 public static Quaternion Lerp(Quaternion from, Quaternion to, float t);

功能说明 此方法用于返回从参数from到to的线性插值。当参数t≤0时返回值为from，当参数t≥1时返回值为to。此方法执行速度比Slerp方法快，一般情况下可代替Slerp方法。

实例演示 下面通过实例演示方法Lerp的使用。

```
using UnityEngine;
using System.Collections;
```

```
public class Slerp_ts : MonoBehaviour
{
    public Transform A, B, C, D;
    float speed = 0.2f;
    //分别演示方法Slerp和Lerp的使用
    void Update()
    {
        C.rotation = Quaternion.Slerp(A.rotation, B.rotation, Time.time * speed);
        D.rotation = Quaternion.Lerp(A.rotation, B.rotation, Time.time * speed);
    }
}
```

在这段代码中，首先声明了4个Transform变量A、B、C和D，分别指向场景中不同的物体对象，然后在Update方法中分别演示了方法Slerp和Lerp的使用，请自行运行程序查看。

8.3.7　LookRotation方法：设置Quaternion的朝向

基本语法　(1) public static Quaternion LookRotation(Vector3 forward);

(2) public static Quaternion LookRotation(Vector3 forward, Vector3 upwards);

其中参数forward为返回Quaternion的forward朝向。

功能说明　此方法用于返回一个Quaternion实例，使GameObject对象的z轴朝向参数forward方向。此方法与方法SetLookRotation (view : Vector3, up : Vector3 = Vector3.up)功能相同，只是用法上有些不同。

实例演示　下面通过实例演示方法LookRotation的使用。

```
using UnityEngine;
using System.Collections;

public class LookRotation_ts : MonoBehaviour
{
    public Transform A, B, C, D;
    Quaternion q1 = Quaternion.identity;

    void Update()
    {
        //使用实例方法
        //不可直接使用C.rotation.SetLookRotation(A.position,B.position);
        q1.SetLookRotation(A.position, B.position);
        C.rotation = q1;
        //使用类方法
        D.rotation = Quaternion.LookRotation(A.position, B.position);
        //绘制直线，请在Scene视图中查看
        Debug.DrawLine(Vector3.zero, A.position, Color.white);
        Debug.DrawLine(Vector3.zero, B.position, Color.white);
        Debug.DrawLine(C.position, C.TransformPoint(Vector3.up * 2.5f), Color.white);
        Debug.DrawLine(C.position, C.TransformPoint(Vector3.forward * 2.5f),
```

```
            Color.white);
        Debug.DrawLine(D.position, D.TransformPoint(Vector3.up * 2.5f), Color.white);
        Debug.DrawLine(D.position, D.TransformPoint(Vector3.forward * 2.5f),
            Color.white);
    }
}
```

在这段代码中，首先声明了4个Transform变量用于指向不同的对象，然后在Update方法中调用方法SetLookRotation，使得C的z轴方向与向量A相同，y轴方向尽量接近向量B。接着调用静态方法LookRotation，使得D的z轴方向与向量A相同，y轴方向尽量接近向量B，如图8-9所示。请自行运行程序在Scene视图而非Game视图中查看。程序运行时可以在Scene视图中试着拖动A或B的位置，来观察C和D的变化。

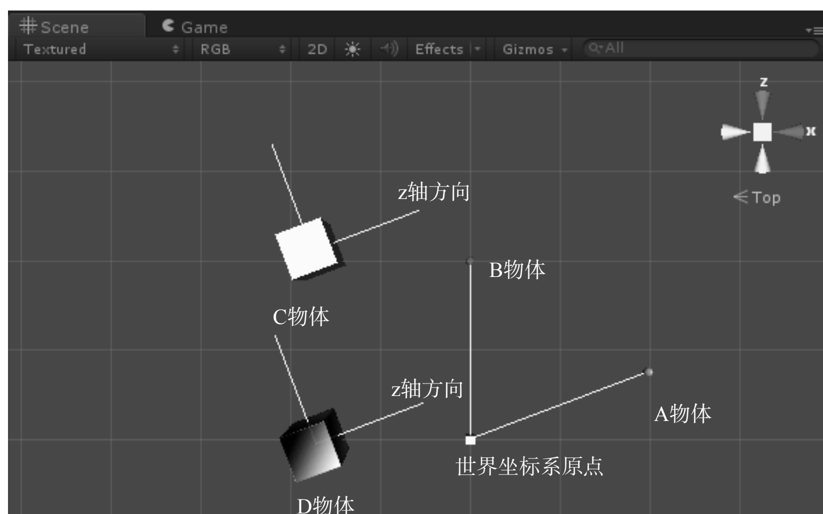


图8-9　实例演示中Scene面板截图

8.3.8　RotateTowards方法：Quaternion插值

基本语法　public static Quaternion RotateTowards(Quaternion from, Quaternion to, float maxDegreesDelta);

其中参数from为起始Quaternion，参数to为结束Quaternion，参数maxDegreesDelta为每帧最大角度值。

功能说明　此方法用于返回从参数from到to的插值，且返回值的最大角度不超过maxDegreesDelta。此方法功能与方法Slerp相似，只是maxDegreesDelta指的是角度值，不是插值系数。当maxDegreesDelta<0时，将沿着从to到from的方向插值计算。

实例演示　下面通过实例演示方法RotateTowards的使用。

```
using UnityEngine;
using System.Collections;

public class RotateTowards_ts : MonoBehaviour {
    public Transform A, B, C;
    float speed = 10.0f;

    void Update()
{
        //调用方法RotateTowards，并将其返回值赋给C.rotation
        C.rotation = Quaternion.RotateTowards(A.rotation, B.rotation, Time.time *
            speed-40.0f);
        Debug.Log("C与A的欧拉角的差值：" + (C.eulerAngles-A.eulerAngles) + " maxDegreesDelta: +"
          (Time.time * speed - 40.0f));
    }
}
```

在这段代码中，首先声明了 3 个变量A、B和C，用于指向场景中不同的物体对象，然后在Update方法中调用方法RotateTowards，使得物体C的rotation随着时间不断接近B的rotation。请自行运行程序查看，图8-10是一张Debug输出截图。

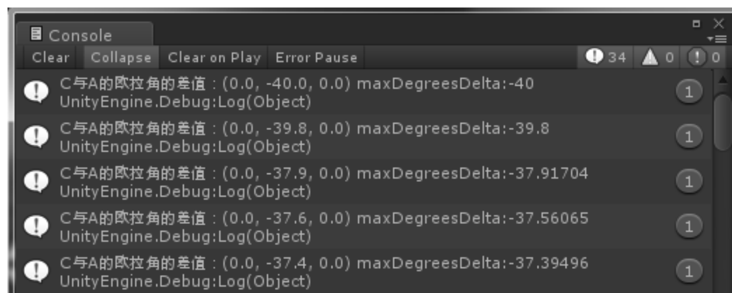


图8-10　RotateTowards实例演示的运行结果

8.3.9　Slerp方法：球面插值

基本语法　public static Quaternion Slerp(Quaternion from, Quaternion to, float t);

功能说明　此方法用于返回从参数from到to的球面插值。当参数t≤0时返回值为from，当参数t≥1时返回值为to。一般情况下可用方法Lerp代替。

实例演示　下面通过实例演示方法Slerp的使用。

```
using UnityEngine;
using System.Collections;

public class Slerp_ts : MonoBehaviour
{
    public Transform A, B, C, D;
    float speed = 0.2f;
    //分别演示方法Slerp和Lerp的使用
```

```
    void Update()
    {
        C.rotation = Quaternion.Slerp(A.rotation, B.rotation, Time.time * speed);
        D.rotation = Quaternion.Lerp(A.rotation, B.rotation, Time.time * speed);
    }
}
```

在这段代码中，首先声明了4个Transform变量A、B、C和D，分别指向场景中不同的物体对象，然后在Update方法中分别演示了方法Slerp和Lerp的使用，请自行运行程序查看。

8.4 Quaternion 类运算符

在Quaternion类中涉及的运算符运算有两个Quaternion实例相乘的运算、一个Quaternion实例和一个Vector3相乘的运算，下面介绍这两种不同的运算。

8.4.1 operator * (lhs : Quaternion, rhs : Quaternion)

功能说明 此运算符用于返回两个Quaternion实例相乘后的结果。设A和B均为GameObject对象的一个实例，有如下代码：

```
B.rotation *= A.rotation;
```

则

- 代码每执行一次，B都会绕着B的局部坐标系的z、x、y轴分别旋转A.eulerAngles.z度、A.eulerAngles.x度和A.eulerAngles.y度，注意它们的旋转次序一定是先饶z轴再绕x轴最后绕y轴进行相应的旋转。另外由于是绕着局部坐标系旋转，故而当绕着其中一个轴旋转时，很可能会影响其余两个坐标轴方向的欧拉角（除非其余两轴的欧拉角都为0才不受影响）。
- 设A的欧拉角为euler_a(ax,ay,az)，则沿着B的初始局部坐标系的euler_a方向向下看，会发现B在绕着euler_a顺时针旋转。B的旋转状况还受其初始状态的影响，可以在运行示例程序时在Inspector面板中更改B的初始欧拉角，从而查看运行状态的不同。
- 此方法主要用于物体自身旋转变换，若想进行自身移动变换，请参考8.4.2节operator * (rotation : Quaternion, point : Vector3)方法。

实例演示 下面通过实例演示两个Quaternion实例的相乘运算。

```
using UnityEngine;
using System.Collections;

public class QxQ_ts : MonoBehaviour {
    public Transform A, B;
    void Start () {
        //设置A的欧拉角
```

```
        //试着更改各个分量查看B的不同旋转状态
        A.eulerAngles = new Vector3(1.0f,1.5f,2.0f);
    }

    void Update () {
        B.rotation *= A.rotation;
        //输出B的欧拉角，注意观察B的欧拉角变化
        Debug.Log(B.eulerAngles);
    }
}
```

在这段代码中，首先声明了两个Transform类型的变量，并在Start方法中对A的欧拉角进行赋值。然后在Update方法中将B.rotation与A.rotation相乘的值赋给B.rotation，从而实现B对象的不断旋转。请自行运行程序查看，图8-11是Debug输出截图，注意观察B的欧拉角的变化，B绕着其自身坐标系的Vector3(1.0f,1.5f,2.0f)方向旋转。虽然每次绕这个轴向旋转的角度相同，但角度的旋转在3个坐标轴上的值都不为零，其中一轴的旋转会影响其他两轴的角度，故而B的欧拉角的各个分量的每次递增值是不固定的。

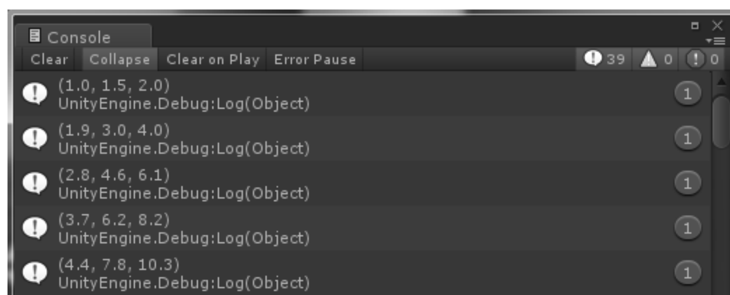


图8-11　实例演示运行结果

8.4.2　operator * (rotation : Quaternion, point : Vector3)

功能说明　此运算符的作用是对参数坐标点point进行rotation变换。例如，设A为Vector3实例，有如下代码：

```
transform.position += transform.rotation *A;
```

则每执行一次代码，transform对应的对象便会沿着自身坐标系中向量A的方向移动A的模长的距离。transform.rotation与A相乘主要来确定移动的方向和距离。

实例演示　下面通过实例演示Quaternion与Vector3的相乘运算。

```
using UnityEngine;
using System.Collections;

public class QxV_ts : MonoBehaviour
{
    public Transform A;
```

```
    float speed = 0.1f;
    //初始化A的position和eulerAngles
    void Start()
    {
        A.position = Vector3.zero;
        A.eulerAngles = new Vector3(0.0f, 45.0f, 0.0f);
    }

    void Update()
    {
        //沿着A自身坐标系的forward方向每帧前进speed距离
        A.position += A.rotation * (Vector3.forward * speed);
        Debug.Log(A.position);
    }
}
```

在这段代码中，首先声明了一个变量A，初始化了一个变量speed，并在Start方法中对变量A的position和eulerAngles初始化。然后在Update方法中，使用Quaternion与Vector3相乘，使得物体A沿着自身坐标系的forward方向每帧前进speed距离。请自行运行程序查看，图8-12是一张Debug输出截图。

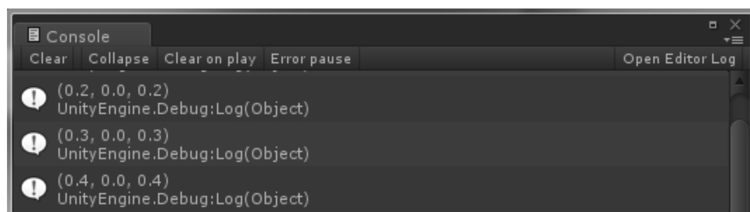


图8-12　实例演示运行结果

8.5　关于 Quaternion 类中相乘运算符的两种重载方式的注解

在Quaternion类中，相乘运算符（*）有两种重载方式，分别为operator *(lhs : Quaternion, rhs : Quaternion)和operator * (rotation : Quaternion, point : Vector3)。

设A为两个Quaternion实例的乘积，结果为Quaternion类型，设B为Quaternion实例和Vector3的乘积，结果为Vector3类型，则二者主要有以下异同。

- A与B的相似处是它们都通过自身坐标系的“相乘”方式来实现在世界坐标系中的变换；
- A主要用来实现transform绕着自身坐标系中的某个轴进行旋转，而B主要用来实现transform沿着自身坐标系的某个方向进行移动；
- B的相乘顺序只有Quaternion*Vector3一种形式，而没有Vector3*Quaternion的形式；
- 由于它们都是相对于自身坐标系进行的旋转或移动，故而当自身坐标系的轴向和世界坐标系的轴向不一致时，它们绕着自身坐标系中某个轴向的变动都会影响到物体在世界坐标系中各个坐标轴的变动。

8

第9章 Random类

Random类是Unity中用于产生随机数的类，不可实例化，只有静态属性和静态方法。本章主要介绍了Random类的一些静态属性。

9.1 Random 类静态属性

在Random类中，涉及的静态属性有insideUnitCircle属性、insideUnitSphere属性、onUnitSphere属性、rotationUniform属性、rotation属性和seed属性。由于属性insideUnitCircle、insideUnitSphere和onUnitSphere功能相近，属性rotationUniform和rotation的功能也相近，于是把它们放到一起介绍，下面详细介绍这些属性。

9.1.1 insideUnitCircle属性：圆内随机点

基本语法 public static Vector2 insideUnitCircle { get; }

功能说明 此属性用于返回一个半径为1的圆内的随机点坐标，返回值为Vector2类型。

提　　示 以下两种属性与此属性的功能相似。

- insideUnitSphere 属性:返回一个半径为 1 的球内的随机点坐标,返回值为 Vector3 类型;
- onUnitSphere 属性：返回一个半径为 1 的球表面的随机点坐标，返回值为 Vector3 类型。

实例演示 下面通过实例演示属性insideUnitCircle、insideUnitSphere和onUnitSphere的使用。

```
using UnityEngine;
using System.Collections;

public class insideUnitCircle_ts : MonoBehaviour
{
    public GameObject go;

    void Start()
```

```
    {
        //每隔0.4秒执行一次use_rotationUniform方法
        InvokeRepeating("use_rotationUniform", 1.0f, 0.4f);
    }

    void use_rotationUniform()
    {
        //在半径为5的圆内随机位置实例化一个GameObject对象
        //Vector2实例转为Vector3时，z轴分量默认为0
        Instantiate(go, Random.insideUnitCircle * 5.0f, Quaternion.identity);
        //在半径为5的球内随机位置实例化一个GameObject对象
        Instantiate(go, Vector3.forward * 15.0f + 5.0f * Random.insideUnitSphere,
            Quaternion.identity);
        //在半径为5的球表面随机位置实例化一个GameObject对象
        Instantiate(go, Vector3.forward * 30.0f + 5.0f * Random.onUnitSphere,
            Quaternion.identity);
    }
}
```

在这段代码中，首先声明了一个公共的GameObject变量go，然后在方法use_rotationUniform中分别使用Random属性insideUnitCircle、insideUnitSphere和onUnitSphere的返回值作为实例化对象的参考位置，最后在Start方法中调用InvokeRepeating方法，每隔0.4秒执行一次use_rotationUniform方法。请自行运行程序查看，图9-1是一张运行时截图。

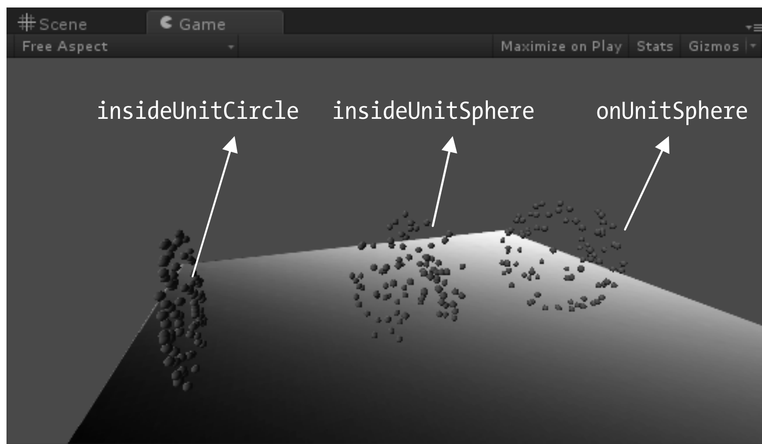


图9-1 实例演示运行截图

9.1.2 rotationUniform属性：均匀分布特征

基本语法 `public static Quaternion rotationUniform { get; }`

功能说明 此属性用于返回一个随机且符合均匀分布特征的rotation值。所谓均匀分布特征，通俗地讲就是指每个可能出现的随机数的概率是相等的。例如，设随机数产生函数

f(x)=x%7，x的取值范围为[0,10]，则当x从0增大到10时，出现f(x)=0的次数有两次，分别为当x=0和x=7时；而f(x)=5的机会只有一次，即x=5时，所以f(x)的序列不是均匀的随机分布。为了使得f(x)中每个值出现的概率相等，可以把x的取值范围扩展为[0,7n)，其中n为正整数，这样f(x)产生每个值的概率就相等了。

提 示 Random类的rotation属性与此属性功能相近，不同之处在于rotation属性只是产生一个随机的rotation值，不符合均匀分布特征，但其计算速度比rotationUniform快，一般情况下可用Random类的rotation属性产生一个随机的rotation值。

实例演示 下面通过实例演示rotationUniform的使用。

```
using UnityEngine;
using System.Collections;

public class rotationUniform_ts : MonoBehaviour
{
    public GameObject go;
    GameObject cb, sp;
    GameObject cb1, sp1;

    void Start()
    {
        //分别获取cb、sp、cb1和sp1对象
        cb = GameObject.Find("Cube");
        sp = GameObject.Find("Cube/Sphere");
        cb1 = GameObject.Find("Cube1");
        sp1 = GameObject.Find("Cube1/Sphere1");
        //每隔0.4秒执行一次use_rotationUniform方法
        InvokeRepeating("use_rotationUniform", 1.0f, 0.4f);
    }

    void use_rotationUniform()
    {
        //使用rotationUniform产生符合均匀分布特征的rotation
        cb.transform.rotation = Random.rotationUniform;
        //使用rotation产生一个随机rotation
        cb1.transform.rotation = Random.rotation;
        //分别在sp和sp1的位置实例化一个GameObject对象
        Instantiate(go, sp.transform.position, Quaternion.identity);
        Instantiate(go, sp1.transform.position, Quaternion.identity);
    }
}
```

在这段代码中，首先声明了一个公共的GameObject变量go和4个私有的GameObject类型变量，并在Start方法中将4个私有变量分别指向场景中不同的对象。接着在方法use_rotationUniform中分别使用属性rotationUniform和rotation参数两个随机的rotation值，并将它们分别赋予cb和cb1。最后分别在sp和sp1的位置实例化一个GameObject对象。请自行运行程序查看。

9.1.3 seed属性：随机数种子

基本语法 public static int seed { get; set; }

功能说明 此属性用来设置随机数的种子。在计算机中产生随机数的方法有很多，但每种方法都需要一个种子，例如经典的伪随机数产生函数：$f(x)=f(x-1)*a+b$，其中a、b为已知的固定数值，那么只要知道某个x对应的f值，就可以推算出所有的值。通常情况下，会把f(0)当做随机数产生的种子，即只要知道了f(0)的值就可以推算出f(1)、f(2)…的值。总之，相同的Random.seed值对应着相同的随机数序列，如果不人为设定其值，Unity会根据某种算法自动产生一个种子。

实例演示 下面通过实例演示属性seed的使用。

```
using UnityEngine;
using System.Collections;

public class Seed_ts : MonoBehaviour
{
    void Start()
    {
        //设置随机数的种子
        //不同的种子产生不同的随机数序列
        //对于相同的种子，在程序每次启动时其序列是相同的
        Random.seed = 1;
    }
    void Update()
    {
        Debug.Log(Random.value);
    }
}
```

在这段代码中，首先在Start方法中设置了随机数的种子，然后在Update方法中打印出每次的随机数值，图9-2是程序在本机上每次输出的序列值。

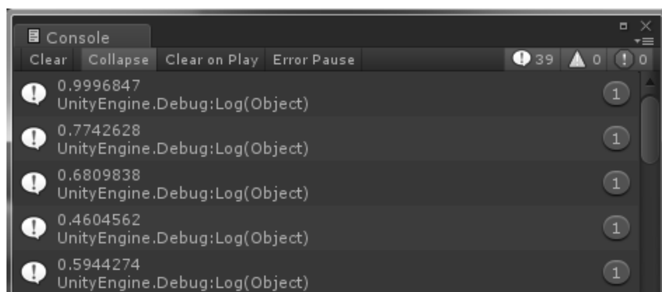


图9-2 seed实例演示的运行结果

9.2　Random 类其他常用静态属性功能简介

本小节简要介绍一些Random类的其他常用静态属性的功能，列举如下。

- insideUnitSphere属性：返回一个半径为1的球内的随机点坐标，返回值为Vector3类型；
- onUnitSphere属性：返回一个半径为1的球表面的随机点坐标，返回值为Vector3类型；
- rotation属性：用于返回一个随机的rotation值，返回值为Quaternion类型；
- value属性：用于返回一个[0.0f,1.0f]区间内的随机数。

第 10 章

Rigidbody类

10

Rigidbody类的功能是用来模拟GameObject对象在现实世界中的物理特性，包括重力、阻力、质量、速度等。对Rigidbody对象属性的赋值代码通常放在脚本的OnFixedUpdate()方法中。本章主要介绍了Rigidbody类的一些实例属性和实例方法，最后对Rigidbody类中功能相近、关联性较强的API之间的关系进行了注解。

10.1 Rigidbody 类实例属性

在Rigidbody类中，涉及的实例属性有collisionDetectionMode、drag、inertiaTensor、mass和velocity，下面详细介绍这些属性。

10.1.1 collisionDetectionMode属性：碰撞检测模式

基本语法 public CollisionDetectionMode collisionDetectionMode { get; set; }

功能说明 此属性用于设置刚体的碰撞检测模式。刚体的碰撞检测模式有3种，即枚举类型CollisionDetectionMode的 3 个值。

- Discrete：静态离散检测模式，为系统的默认设置。在此模式下，只有在某一帧中两物体的碰撞器发生重叠时才能被检测到，这样就有可能导致某物体的前一帧在另一个刚体的上方，而下一帧移动到了另一个刚体的下方，这样就会发生穿越现象。
- Continuous：静态连续检测模式，一般用在高速运动刚体的目标碰撞体上，防止被穿越，检测强度比Discreet强。
- ContinuousDynamic：最强的连续动态检测模式，一般用在两个高速运动的物体上，防止互相穿越。其计算消耗最大，一般情况下慎用。

总之，无论哪种检测模式都有可能被穿越，为了防止穿越现象的发生，除了设置其碰撞检测模式外，还要适当增加两物体碰撞器的厚度，一般不要小于0.1，同时尽量降低两物体碰撞时的相对速度。

实例演示 下面通过实例演示属性collisionDetectionMode的使用。

```
using UnityEngine;
using System.Collections;

public class CollisionDetectionMode_ts : MonoBehaviour
{
    public Rigidbody A, B;
    Vector3 v1, v2;
    void Start()
    {
        A.useGravity = false;
        B.useGravity = false;
        v1 = A.position;
        v2 = B.position;
    }
    void OnGUI()
    {
        if (GUI.Button(new Rect(10.0f, 10.0f, 200.0f, 45.0f), "Discrete模式不被穿越"))
        {
            inists();
            A.collisionDetectionMode = CollisionDetectionMode.Discrete;
            B.collisionDetectionMode = CollisionDetectionMode.Discrete;
            A.velocity = new Vector3(0.0f,-10.0f,0.0f);
        }
        if (GUI.Button(new Rect(10.0f, 60.0f, 200.0f, 45.0f), "Discrete模式被穿越"))
        {
            inists();
            A.collisionDetectionMode = CollisionDetectionMode.Discrete;
            B.collisionDetectionMode = CollisionDetectionMode.Discrete;
            A.velocity = new Vector3(0.0f, -40.0f, 0.0f);
        }
        if (GUI.Button(new Rect(10.0f, 110.0f, 200.0f, 45.0f), "Continuous模式不被穿越"))
        {
            inists();
            A.collisionDetectionMode = CollisionDetectionMode.Continuous;
            B.collisionDetectionMode = CollisionDetectionMode.Continuous;
            A.velocity = new Vector3(0.0f, -20.0f, 0.0f);
        }
        if (GUI.Button(new Rect(10.0f, 160.0f, 200.0f, 45.0f), "Continuous模式被穿越"))
        {
            inists();
            A.collisionDetectionMode = CollisionDetectionMode.Continuous;
            B.collisionDetectionMode = CollisionDetectionMode.Continuous;
            A.velocity = new Vector3(0.0f, -15.0f, 0.0f);
            B.velocity = new Vector3(0.0f, 15.0f, 0.0f);
        }
        if (GUI.Button(new Rect(10.0f, 210.0f, 200.0f, 45.0f), "ContinuousDynamic模式"))
        {
            inists();
            A.collisionDetectionMode = CollisionDetectionMode.ContinuousDynamic;
            B.collisionDetectionMode = CollisionDetectionMode.ContinuousDynamic;
            A.velocity = new Vector3(0.0f, -200.0f, 0.0f);
            B.velocity = new Vector3(0.0f, 200.0f, 0.0f);
        }
        if (GUI.Button(new Rect(10.0f, 260.0f, 200.0f, 45.0f), "重置"))
```

```
        {
            inists();
        }
    }
    //初始化A、B
    void inists() {
        A.position = v1;
        A.rotation = Quaternion.identity;
        A.velocity = Vector3.zero;
        A.angularVelocity = Vector3.zero;
        B.position = v2;
        B.rotation = Quaternion.identity;
        B.velocity = Vector3.zero;
        B.angularVelocity = Vector3.zero;
    }
}
```

在这段代码中，首先声明了两个Rigidbody变量A、B和两个Vector3变量v1、v2，然后在Start方法中设置A、B的useGravity属性为false，并将A、B的Position赋值给v1和v2，然后在OnGUI方法中定义了多个Button，用来演示Discrete、Continuous和ContinuousDynamic的功能。最后定义了一个inists方法，用于重置变量A、B对应刚体的状态。请自行运行程序查看。

10.1.2 drag属性：刚体阻力

基本语法 public float drag { get; set; }

功能说明 此属性用于给刚体添加一个阻力。drag值越大刚体速度减慢得越快，当drag>0时，刚体在增加到一定速度后会匀速移动。

提　　示 刚体在自由落体运动中的最大速度值只与Gravity和drag值有关，与质量Mass无关。

实例演示 下面通过实例演示属性drag的使用。

```
using UnityEngine;
using System.Collections;

public class Drag_ts : MonoBehaviour
{
    public Rigidbody R;
    float drags = 20.0f;
    string str = "200";
    //初始化R.drag
    void Start()
    {
        R.drag = drags;
    }

    void OnGUI()
```

```
    {
        GUI.Label(new Rect(10.0f, 10.0f, 200.0f, 45.0f), "Gravity:" + Physics.gravity);
        str = (GUI.TextField(new Rect(10.0f, 60.0f, 200.0f, 45.0f), str));
        if (GUI.Button(new Rect(10.0f, 210.0f, 200.0f, 45.0f), "compute"))
        {
            drags = float.Parse(str);
            R.drag = drags;
        }
        GUI.Label(new Rect(10.0f, 110.0f, 200.0f, 45.0f), "Velocity:" + R.velocity.y);
        GUI.Label(new Rect(10.0f, 160.0f, 200.0f, 45.0f), "drag:" + drags);
    }
}
```

在这段代码中，首先声明了一个Rigidbody变量R，并在Start方法中对其初始化，然后在OnGUI方法中绘制了一些GUI组件，用于测试不同的drag值下，物体下落的最终速度。请自行运行程序查看，下表是在本机上进行的一组测试结果。

表10-1　drag实例演示本机的一组测试结果

drag	0.1	0.2	0.3	0.4	0.5
max_v	97.9	48.9	32.5	24.3	19.42
drag	0.6	0.7	0.8	0.9	1
max_v	16.15	13.82	12.1	10.7	9.61
drag	1.5	2	5	10	20
max_v	6.34	4.7	1.76	0.78	0.29

10.1.3　inertiaTensor属性：惯性张量

功能说明　此属性用于设置刚体的惯性张量。在距离重心同等的条件下，刚体会向张量值大的一边倾斜。例如，设有如下代码：

```
rigidbody. inertiaTensor=new Vector3(5.0f,10.0f,1.0f);
```

起始状态如图10-1所示，则刚体会向y轴所在的方向倾斜翻转，如图10-2所示。

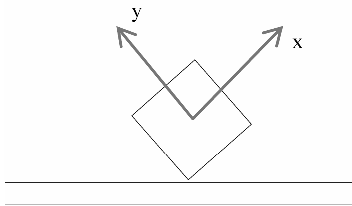


图10-1　起始物体状态

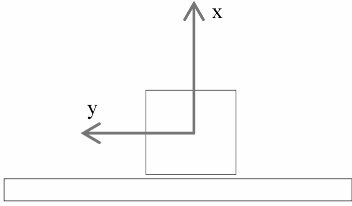


图10-2　最终物体状态

实例演示　下面通过实例演示属性inertiaTensor的使用。

```
using UnityEngine;
using System.Collections;
```

```
public class inertiaTensor_ts : MonoBehaviour
{
    void OnGUI()
    {
        if (GUI.Button(new Rect(10.0f, 10.0f, 160.0f, 45.0f), "x轴惯性张量大于y轴"))
        {
            transform.position = new Vector3(0, 4, 0);
            //transform绕z轴旋转45度
            transform.rotation = Quaternion.Euler(0.0f, 0.0f, 45.0f);
            //设置rigidbody的惯性张量
            //x轴分量值大于y轴，则刚体会向x轴方向倾斜
            rigidbody.inertiaTensor = new Vector3(15.0f, 10.0f, 1.0f);
        }
        if (GUI.Button(new Rect(10.0f, 60.0f, 160.0f, 45.0f), "y轴惯性张量大于x轴"))
        {
            transform.position = new Vector3(0, 4, 0);
            transform.rotation = Quaternion.Euler(0.0f, 0.0f, 45.0f);
            //设置rigidbody的惯性张量
            //x轴分量值小于y轴，则刚体会向y轴方向倾斜
            rigidbody.inertiaTensor = new Vector3(5.0f, 10.0f, 1.0f);
        }
        if (GUI.Button(new Rect(10.0f, 110.0f, 160.0f, 45.0f), "x轴和y轴惯性张量相同"))
        {
            transform.position = new Vector3(0, 4, 0);
            transform.rotation = Quaternion.Euler(0.0f, 0.0f, 45.0f);
            //设置rigidbody的惯性张量
            //x轴和y轴惯性张量相同，则刚体会保持静止
            rigidbody.inertiaTensor = new Vector3(10.0f, 10.0f, 1.0f);
        }
    }
}
```

在这段代码的OnGUI方法中，分别定义了3个Button，用来模拟不同惯性张量情况下刚体的运动情况，请参考代码注释，自行运行程序查看运行结果。

10.1.4 mass属性：刚体质量

基本语法 public float mass { get; set; }

功能说明 此属性用于设置或返回刚体的质量。一般刚体质量取值在0.1附近模拟最佳，最大不要超过10，否则容易出现模拟不稳定的情况。此属性使用情况说明如下。

- 对于自由落体运动，物体的速度只与重力加速度Gravity和空气阻力drag有关，与质量mass无关。
- mass的主要作用是在物体发生碰撞时计算碰撞后物体的速度。当一个物体分别去撞击mass大的物体和mass小的物体时，根据动量守恒定律，较重的物体被撞后的速度要慢于较轻的物体。

实例演示 下面通过实例演示属性mass的使用。

```
using UnityEngine;
using System.Collections;

public class Mass_ts : MonoBehaviour
{
    public Rigidbody r1, r2, r3, r4, r5;
    Vector3 v3 = Vector3.zero;
    void Start()
    {
        //r1和r2质量不同，但它们的速度始终相同
        //r4和r5质量不同，当r3以同样的速度撞r4和r5后速度明显不同
        r1.mass = 0.1f;
        r2.mass = 5.0f;
        r3.mass = 2.0f;
        r4.mass = 0.1f;
        r5.mass = 4.0f;

        r3.useGravity = false;
        r4.useGravity = false;
        r5.useGravity = false;
        v3 = r3.position;
    }

    void FixedUpdate()
    {
        Debug.Log(Time.time + " R1的速度：" + r1.velocity);
        Debug.Log(Time.time + " R2的速度：" + r2.velocity);
        Debug.Log(Time.time + " R3的速度：" + r3.velocity);
        Debug.Log(Time.time + " R4的速度：" + r4.velocity);
        Debug.Log(Time.time + " R5的速度：" + r5.velocity);
    }

    void OnGUI()
    {
        if (GUI.Button(new Rect(10.0f, 10.0f, 200.0f, 45.0f), "用R3撞R4"))
        {
            r3.position = v3;
            r3.rotation = Quaternion.identity;
            r3.velocity = new Vector3(4.0f, 0.0f, 0.0f);
        }

        if (GUI.Button(new Rect(10.0f, 60.0f, 200.0f, 45.0f), "用R3撞R5"))
        {
            r3.position = v3;
            r3.rotation = Quaternion.identity;
            r3.velocity = new Vector3(0.0f, 0.0f, 4.0f);
        }
    }
}
```

在这段代码中，首先声明了5个不同的Rigidbody变量，然后在Start方法中分别设置其mass属性和useGravity属性（useGravity属性默认为true），然后在OnGUI方法中定义

了两个不同功能的Button，最后在FixedUpdate方法中打印出每帧各个刚体的移动速度。请自行运行程序查看，运行程序后你会发现，刚体r1和r2质量虽然不同，但速度却始终相同；当r3以同样的速度分别去撞r4和r5后，r4和r5的速度明显不同。

10.1.5 velocity属性：刚体速度

基本语法 public Vector3 velocity { get; set; }

功能说明 此属性用于设置或返回刚体的速度值，其使用说明如下。

- ❑ 在脚本中无论是给刚体赋予一个Vector3类型的速度向量v，还是获取当前刚体的速度v，v的方向都是相对世界坐标系而言的。
- ❑ velocity的单位是米每秒，而不是帧每秒，其中米是Unity中默认的长度单位。

实例演示 下面通过实例演示属性velocity的使用。

```
using UnityEngine;
using System.Collections;

public class Velocity_ts : MonoBehaviour {
    public Rigidbody r1,r2;
    // Use this for initialization
    void Start () {
        //给父物体r1一个向-z轴的速度，给子物体一个+z轴的速度
        r1.velocity = new Vector3(0.0f,0.0f,-15.0f);
        r2.velocity = new Vector3(0.0f, 0.0f, 10.0f);
    }

    void OnGUI() {
        GUI.Label(new Rect(10.0f,8.0f,300.0f,40.0f),"R1当前速度："+r1.velocity);
        GUI.Label(new Rect(10.0f,58.0f,300.0f,40.0f),"R2当前速度："+r2.velocity);
        Debug.Log("R1当前速度：" + r1.velocity);
        Debug.Log("R2当前速度：" + r2.velocity);
    }
}
```

在这段代码中，首先声明了两个Rigidbody类型的变量r1和r2，然后在Start方法中对r1和r2分别赋予不同的速度，最后在OnGUI方法中打印出r1和r2的速度值，结果如图10-3所示。

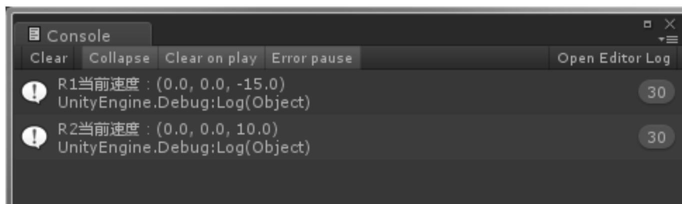


图10-3 velocity实例演示的运行结果

10.2　Rigidbody 类实例方法

在Rigidbody类中涉及的实例方法有AddExplosionForce方法、AddForceAtPosition方法、AddTorque方法、ClosestPointOnBounds方法、GetPointVelocity方法、MovePosition方法、Sleep方法、SweepTest方法、SweepTestAll方法和WakeUp方法，下面详细介绍这些方法。

10.2.1　AddExplosionForce方法：模拟爆炸力

基本语法　(1) `public void AddExplosionForce(float explosionForce, Vector3 explosionPosition, float explosionRadius);`

(2) `public void AddExplosionForce(float explosionForce, Vector3 explosionPosition, float explosionRadius, float upwardsModifier);`

(3) `public void AddExplosionForce(float explosionForce, Vector3 explosionPosition, float explosionRadius, float upwardsModifier, ForceMode mode);`

其中参数explosionForce为爆炸点施加的力的大小，参数explosionPosition为爆炸点坐标（相对世界坐标系），参数explosionRadius为爆炸作用力有效半径，参数upwardsModifier为爆炸力作用点在y轴方向上的偏移，参数mode为爆炸力的作用模式，默认为ForceMode.Force。

功能说明　此方法用于对刚体添加一个模拟爆炸效果的作用力。设爆炸力大小为F，爆炸点坐标为E，有效半径为R，y轴的偏离量为y_m，刚体A的坐标为P，A受到的爆炸作用力为：

```
A. AddExplosionForce(F,E,R,y_m);
```

则可以得出以下结论。

- 爆炸点E作用到A上的力的大小由E点到A表面的最近距离决定。如图10-4所示，尽管E到刚体A的空间距离为模长|EP|，但爆炸力的作用距离却是E到A的左下角B的距离|EB|，即爆炸力作用到A的大小为：

$$FA = \frac{R - |EB|}{R} \times F$$

  当|EB|>R时，FA=0。由上可知，有可能刚体A发生了移动，但作用到A上的力的大小却没有改变，如图10-5所示，当刚体从粗实线的位置移动到细实线再移动到虚线的位置时，爆炸点到刚体的有效作用力距离始终是|Ep|。

- 爆炸力作用在A上的方向由爆炸点E、刚体A的位置P和y轴偏移量y_m共同决定。
  - 当y_m为默认值0时，作用力的方向即为从爆炸点到刚体A最近表面的方向，如图10-4所示的EB方向。

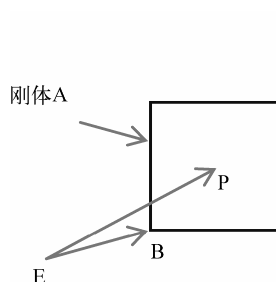


图10-4 爆炸点与物体位置示意图

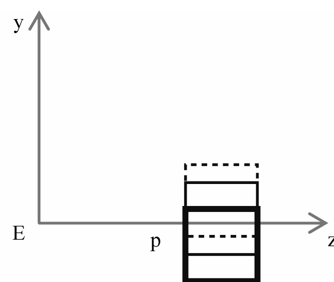


图10-5 物体的垂直移动

- 当y_m<0时，作用点向y轴正向移动y_m，如图10-6中从e1点移动到e2点，然后再以e2为爆炸原点求e2到刚体的作用力方向，如图10- 6 中从e2到B的方向。需要注意的是，无论爆炸点被偏移到什么地方，作用于刚体的力的大小都是由e1点到刚体的最短距离决定的，与偏移后的爆炸点位置无关。
- 当y_m>0时，作用点向y轴负向移动y_m，如图10-7中从e1点移动到e2点，然后再以e2为爆炸原点求e2到刚体的作用力方向，如图10-7中从e2到B的方向。需要注意的是，无论爆炸点被偏移到什么地方，作用于刚体的力的大小都是由e1点到刚体的最短距离决定的，与偏移后的爆炸点位置无关。

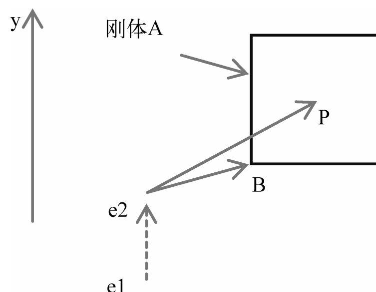


图10-6 作用点向y轴正向移动

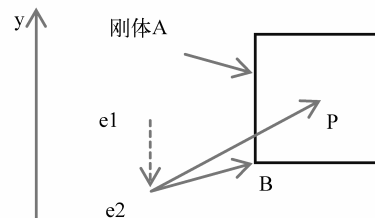


图10-7 作用点向y轴负向移动

综上可知：

- ❑ 爆炸力作用在刚体上的力的大小和方向是分开计算的；
- ❑ 当爆炸点E固定时，刚体在某个范围移动时受到的爆炸力的大小可能不变；
- ❑ 当作用力半径R=0时，所有接受爆炸点E作用的刚体受到的作用力大小都为F。

实例演示 下面通过实例演示方法AddExplosionForce的使用。

```
using UnityEngine;
using System.Collections;

public class AddExplosionForce_ts : MonoBehaviour
{
```

```
public Rigidbody A;
public Transform Z;//Scene视图中显示爆炸点坐标
Vector3 E = Vector3.zero;//爆炸点坐标
float F, R, y_m;
bool is_change = false;

void Start()
{
    //初始位置使得爆炸点和A的x、y轴坐标值相等
    //可以更改F及R的大小查看运行时效果
    E = A.position - new Vector3(0.0f, 0.0f, 3.0f);
    F = 40.0f;
    R = 10.0f;
    y_m = 0.0f;
    A.transform.localScale = Vector3.one * 2.0f;
    Z.position = E;
}

void FixedUpdate()
{
    if (is_change)
    {
        A.AddExplosionForce(F, E, R, y_m);
        is_change = false;
    }
}

void OnGUI()
{
    //当爆炸点和A的重心的两个坐标轴的值相等时，A将平移不旋转
    if (GUI.Button(new Rect(10.0f, 10.0f, 200.0f, 45.0f), "刚体移动不旋转"))
    {
        is_change = true;
        inits();
    }
    //虽然受力大小不变，但产生扭矩发生旋转
    if (GUI.Button(new Rect(10.0f, 60.0f, 200.0f, 45.0f), "刚体发生移动但受力大小不变"))
    {
        inits();
        A.position += new Vector3(0.5f, -0.5f, 0.0f);
        is_change = true;
    }
    if (GUI.Button(new Rect(10.0f, 110.0f, 200.0f, 45.0f), "按最近表面距离计算力的大小"))
    {
        inits();
        A.position += new Vector3(0.0f, 2.0f, 0.0f);
        is_change = true;
    }
    //y轴的偏移改变了A的原始方向
    //可以更改y_m的值查看不同的效果
    if (GUI.Button(new Rect(10.0f, 160.0f, 200.0f, 45.0f), "y轴发生偏移"))
    {
        inits();
        is_change = true;
```

```
            A.position += new Vector3(0.0f, 2.0f, 0.0f);
            y_m = -2.0f;
        }
    }
    //初始化数据
    void inits()
    {
        A.velocity = Vector3.zero;
        A.angularVelocity = Vector3.zero;
        A.position = E + new Vector3(0.0f, 0.0f, 3.0f);
        A.transform.rotation = Quaternion.identity;
        y_m = 0.0f;
    }
}
```

在这段代码中，首先声明了一个Rigidbody变量A和一些其他变量，并在Start方法中对这些变量进行初始化，然后在OnGUI方法中定义了多个功能不同的Button，用于演示方法AddExplosionForce在不同条件下的作用情况，如代码注释所述，最后在FixedUpdate方法中控制方法AddExplosionForce的调用，请自行运行程序查看。

10.2.2 AddForceAtPosition方法：增加刚体点作用力

基本语法 (1) public void AddForceAtPosition(Vector3 force, Vector3 position);

(2) public void AddForceAtPosition(Vector3 force, Vector3 position, ForceMode mode);

其中参数force为扭矩向量，参数position为作用点坐标值，参数mode为力的作用方式。

功能说明 此方法用于为参数position点增加一个力force，其参考坐标系为世界坐标系，作用方式为mode，默认值为ForceMode.Force。此方法与方法AddForce不同，AddForce方法对刚体施加力时不会产生扭矩使物体发生旋转，而AddForceAtPosition方法是在某个坐标点对刚体施加力，这样很可能会产生扭矩使刚体产生旋转，具体如下。

- 当力的作用点在刚体重心时，刚体不发生旋转；
- 当力的作用点不在刚体重心时，由于作用点的扭矩会使刚体发生旋转，但是，当作用力的方向经过刚体的重心坐标时不发生旋转。

实例演示 下面通过实例演示方法AddForceAtPosition的使用。

```
using UnityEngine;
using System.Collections;

public class AddForceAtPosition_ts : MonoBehaviour
{
    public Rigidbody A, B, C;
    Vector3 m_force = new Vector3(0.0f, 0.0f, 10.0f);

    void FixedUpdate()
    {
```

10

```
        //当力的作用点在刚体重心时，刚体不发生旋转
        A.AddForceAtPosition(m_force, A.transform.position, ForceMode.Force);
        //当力的作用点不在刚体重心时，由于作用点的扭矩会使刚体发生旋转
        B.AddForceAtPosition(m_force, B.transform.position + new Vector3(0.0f, 0.3f, 0.0f),
            ForceMode.Force);
        //但是，当力的作用点和刚体重心坐标的差向量与作用力的方向同向时不发生旋转
        C.AddForceAtPosition(m_force, C.transform.position + new Vector3(0.0f, 0.0f, 0.3f),
            ForceMode.Force);
        Debug.Log("A的欧拉角：" + A.transform.eulerAngles);
        Debug.Log("B的欧拉角：" + B.transform.eulerAngles);
        Debug.Log("C的欧拉角：" + C.transform.eulerAngles);
    }
}
```

在这段代码中，首先声明了 3 个Rigidbody变量和一个Vector3变量，然后在方法FixedUpdate中分别对刚体A、B、C施加作用力，最后打印出刚体A、B、C的欧拉角，如图10-8所示。由输出结果可知，只有刚体B发生了旋转，刚体A和C的角度未发生变化，正如代码注释所述。

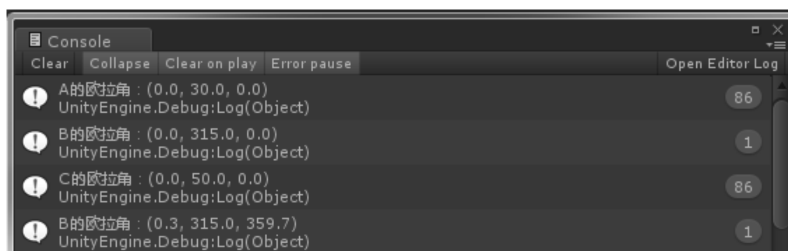


图10-8 AddForceAtPosition实例演示的运行结果

10.2.3 AddTorque方法：刚体添加扭矩

基本语法 (1) public void AddTorque(Vector3 torque);

(2) public void AddTorque(Vector3 torque, ForceMode mode);

(3) public void AddTorque(float x, float y, float z);

(4) public void AddTorque(float x, float y, float z, ForceMode mode)。

其中参数torque为扭矩向量，参数mode为力的作用方式。

功能说明 此方法用于给刚体添加一个扭矩torque，扭矩的作用力方式由mode决定，默认为ForceMode.Force。例如，设A为立方体刚体，其边长L为2，质量m为1，现在对其施加一个扭矩M=new vector3(0.0f,10.0f,0.0f)，使其y轴转动，由公式

$$M = I \cdot \frac{dw}{dt}$$

和立方体的转动惯量公式

$$I = \frac{mL^2}{6}$$

可得：

$$10 = \frac{1 \times 2^2}{6} \times \frac{dw}{0.02}$$

从而可得dw=0.3，即刚体每帧转动的角度为0.3度，此处mode方式为ForceMode.Force。

提　示 不同形状的刚体及不同的转轴，其转动惯量I的计算方式是不同的，具体请参考其他资料。

实例演示 下面通过实例演示方法AddTorque的使用。

```
using UnityEngine;
using System.Collections;

public class AddTorque_ts : MonoBehaviour {
    public Rigidbody R;
    Vector3 m_torque = new Vector3(0.0f,10.0f,0.0f);
    void Start () {
        R.transform.localScale = new Vector3(2.0f,2.0f,2.0f);
        R.mass = 1.0f;
        R.angularDrag = 0.0f;
        Debug.Log("刚体默认的最大角速度："+R.maxAngularVelocity);
        //可以使用如下代码更改刚体的最大角速度
        //R.maxAngularVelocity = 10.0f;
  }

  void FixedUpdate () {
        //每帧给物体添加一个扭矩，使其转速不断加快
        R.AddTorque(m_torque,ForceMode.Force);
        Debug.Log("刚体当前角速度："+R.angularVelocity);
  }
}
```

在这段代码中，首先声明了一个Rigidbody变量R和一个Vector3变量m_torque,然后在Start方法中对变量R对应的刚体进行初始化设置，最后在方法FixedUpdate方法中给刚体R添加一个扭矩，使得R的转速不断加快，并打印出每帧R的旋转角度，如图10-9所示，具体计算方法请参考功能说明。

提　示 运行程序几秒后，刚体角速度会保持在(0.0,7.0,0.0)而不再增加，这是因为刚体有最大角速度的限制，默认值为7.0f，可以通过属性maxAngularVelocity更改刚体的最大角速度，如代码注释所示。

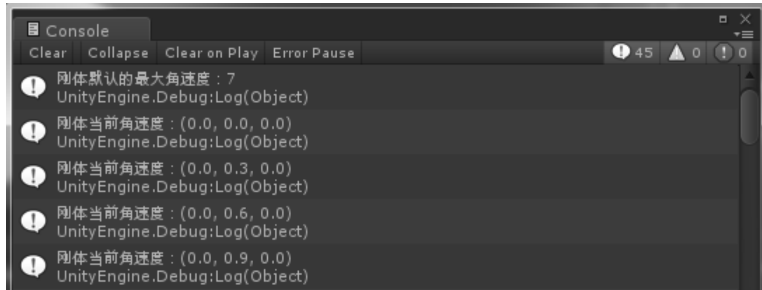


图10-9　AddRelativeForce实例演示的运行结果

10.2.4　ClosestPointOnBounds方法：爆炸点到刚体最短距离

基本语法　public Vector3 ClosestPointOnBounds(Vector3 position);

其中参数position为爆炸点坐标。

功能说明　此方法用于求爆炸点到刚体Collider表面的作用点。如图10-10中所示，爆炸点E到刚体A的最短距离即为向量EB的长度而不是向量EP的长度。此方法通常用在AddExplosionForce中计算爆炸力的大小。

提　　示　返回值为刚体Collider表面上的某一点，而不是Mesh上的点。

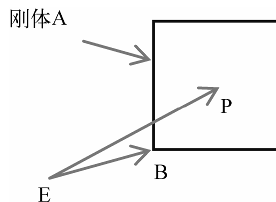


图10-10　爆炸点与物体位置示意图

实例演示　下面通过实例演示方法ClosestPointOnBounds的使用。在本实例中使用的刚体为一个球体Sphere，但在Inspector面板中要将其Collider的Radius改为1，使之与物体的Mesh不吻合，如图10-11和图10-12所示。

```
using UnityEngine;
using System.Collections;

public class ClosestPointOnBounds_ts : MonoBehaviour
{
    public Rigidbody r;
    void Start()
    {
        r.position = new Vector3(0.0f,4.0f,0.0f);
```

```
        Debug.Log(r.ClosestPointOnBounds(new Vector3(5.0f, 4.0f, 0.0f)));
    }
}
```

在这段代码中，首先声明了一个指向场景中的Rigidbody变量r，然后在Start方法中对刚体r的位置进行重置，并打印出了方法ClosestPointOnBounds的返回值，如图10-13所示。



图10-11 更改Collider的Radius

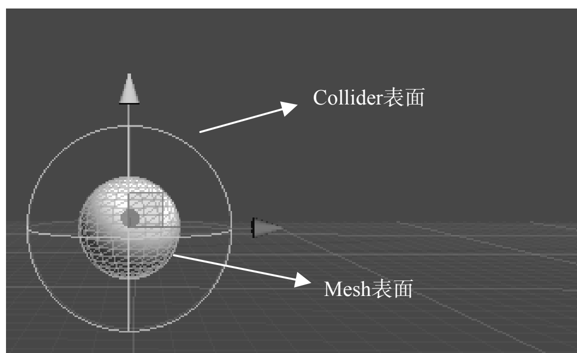


图10-12 Sphere的Collider表面和Mesh表面

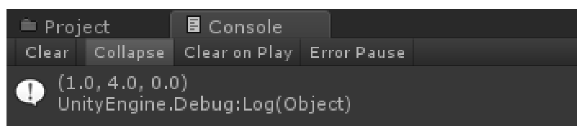


图10-13 ClosestPointOnBounds方法实例演示运行结果

10.2.5 GetPointVelocity方法：刚体点速度

基本语法 public Vector3 GetPointVelocity(Vector3 worldPoint);

其中参数worldPoint为世界坐标系中的点坐标。

功能说明 此方法用于获取世界坐标系中worldPoint点在刚体局部坐标系中的速度。速度的计算会受刚体角速度的影响，对其使用说明如下。

此方法的功能是计算世界坐标系中某一点在刚体局部坐标系中对应位置的速度。此点

10

的坐标可以是世界坐标系中任意点的坐标，可以是刚体表面上某一点对应的世界坐标系的值，也可以不在其表面上。当刚体角速度为Vector3.zero而移动速度不为Vector3.zero时，世界坐标系上任意一点的移动速度都和刚体自身坐标系的原点（即刚体重心）的移动速度相等。而当刚体角速度不为Vector3.zero时，此点的速度值除了计算刚体重心的移动速度，还要计算此点绕刚体重心的角速度。

实例演示 请参考方法GetRelativePointVelocity的实例演示。

10.2.6 GetRelativePointVelocity方法：刚体点相对速度

基本语法 public Vector3 GetRelativePointVelocity(Vector3 relativePoint);

其中参数relativePoint为刚体自身坐标系中的点坐标。

功能说明 此方法用于获取刚体自身坐标系中relativePoint点的速度，速度的计算会受刚体角速度的影响，其使用方法说明如下。

- 此方法的功能是计算刚体自身坐标系中某一点的速度，此坐标点指的是刚体自身坐标系中的任意坐标点，可以是刚体表面上的某一点，也可以不在其表面上。当刚体角速度为Vector3.zero而移动速度不为Vector3.zero时，刚体自身坐标系上任意一点的移动速度都和刚体自身坐标系的原点（即刚体重心）的移动速度相等。而当刚体角速度不为Vector3.zero时，则刚体上任意一点的速度值除了计算刚体重心的移动速度外，还要计算此点绕刚体重心的角速度。
- 此方法和GetPointVelocity作用类似，只是GetPointVelocity方法中使用世界坐标系中的坐标来确定位置，而此方法使用自身坐标系中的坐标来确定位置。为方便理解两个方法的功能，可以把GetPointVelocity中的worldPoint点或GetRelativePointVelocity中的relativePoint点理解为刚体的子类物体的重心坐标值，这样当父类只移动不旋转时，其子类的速度和父类相同；而当父类移动的同时还进行自身旋转时，尽管父类重心依旧朝某一个方向移动，但其子类在跟随父类向前移动时，还要绕着父类重心旋转。

实例演示 下面通过实例演示方法GetRelativePointVelocity的使用。

```
using UnityEngine;
using System.Collections;

public class GetRelativePointVelocity_ts : MonoBehaviour
{
    public Rigidbody A;
    string str = "";
    void Start()
    {
        //给A施加一帧的力，使其产生速度
        A.AddForce(Vector3.forward * 100.0f);
```

```
    }
    void FixedUpdate()
    {
        //A.transform.TransformPoint(Vector3.forward):获取A的局部坐标系中Vector3.forward点
          在世界坐标系中的坐标值
        //这样A.GetPointVelocity(A.transform.TransformPoint(Vector3.forward))和
        //A.GetRelativePointVelocity(Vector3.forward)返回的是同一点的速度，即它们的返回值应
          该相等
        Debug.Log(str + "GetPointVelocity: " + A.GetPointVelocity(A.transform.TransformPoint
            (Vector3.forward)));
        Debug.Log(str + "GetRelativePointVelocity: " + A.GetRelativePointVelocity(Vector3.
            forward));
    }
    void OnGUI()
    {
        //当刚体角速度为Vector3.zero时，任何点的速度都和刚体速度相等
        //当刚体角速度不为Vector3.zero时，坐标系中各个点的速度是不等的
        //GetPointVelocity和GetRelativePointVelocity只是两种计算坐标点的不同方法
        if (GUI.Button(new Rect(10.0f, 10.0f, 200.0f, 45.0f), "增加角速度"))
        {
            A.angularVelocity = new Vector3(0.0f, 45.0f, 0.0f);
            str = "增加角速度后，";
        }
    }
}
```

在这段代码中，首先声明了一个Rigidbody变量A，并在Start方法中给A施加一帧的作用力，使其速度不为零，此时A只有移动速度，没有旋转速度。然后在OnGUI方法中定义一个Button，用来给A增加一个角速度。最后在方法FixedUpdate中分别调用方法GetPointVelocity和GetRelativePointVelocity，打印出每帧中同一坐标点的速度，具体请参见代码所述。图10-14为程序运行结果截图，对结果的具体解释请查看图中注释。

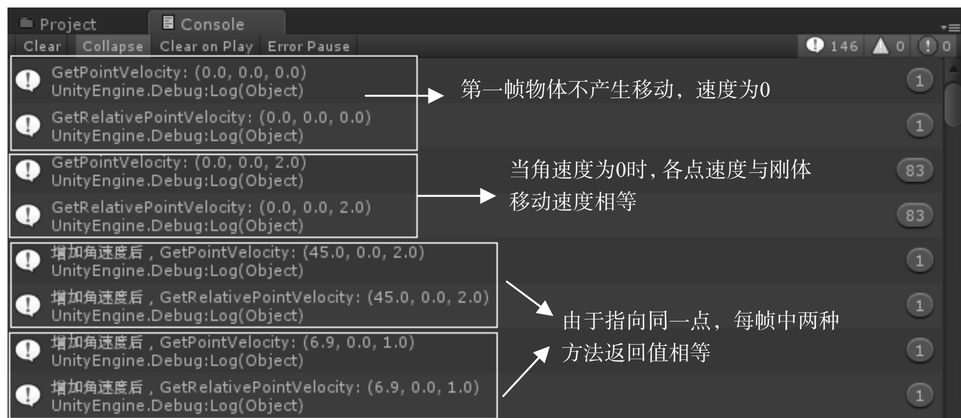


图10-14　GetRelativePointVelocity实例演示的运行结果

10.2.7　MovePosition方法：刚体位置移动

基本语法　public void MovePosition(Vector3 position);

其中参数position为刚体组件要移动到的位置坐标。

功能说明　此方法用于对刚体的位置进行移动，通常用在刚体失去动力学模拟的情况下，即isKinematic为true时。

实例演示　下面通过实例演示方法MovePosition的使用。

```
using UnityEngine;
using System.Collections;

public class MovePOrR_ts : MonoBehaviour
{
    public Rigidbody A;
    bool is_pr = false;
    void Start()
    {
        A.velocity = Vector3.forward * 20.0f;
        A.angularVelocity = Vector3.up * 90.0f;
        //当isKinematic == true时刚体的动力学模拟将失效
        //此时可以使用MovePosition和MoveRotation对刚体进行移动和旋转
        A.isKinematic = true;
    }
    void FixedUpdate()
    {
        if (is_pr)
        {
            A.MovePosition(A.position + Vector3.forward * Time.deltaTime);
            Quaternion deltaRotation = Quaternion.Euler(Vector3.up * 90.0f * Time.deltaTime);
            A.MoveRotation(A.rotation * deltaRotation);
        }
    }
    void OnGUI()
    {
        if (GUI.Button(new Rect(10.0f, 10.0f, 200.0f, 45.0f), "刚体移动与旋转"))
        {
            is_pr = true;
        }
    }
}
```

在这段代码中，首先声明了一个Rigidbody变量A，然后在Start方法中对A的velocity属性和angularVelocity属性进行设置，并关闭刚体A的动力学模拟，即设置isKinematic为true。最后在FixedUpdate方法中调用方法MovePosition和MoveRotation，对刚体A进行移动和旋转，请自行运行程序查看。

10.2.8 Sleep方法：刚体休眠

基本语法 public void Sleep();

功能说明 此方法可使刚体进入休眠状态，且至少休眠一帧。

实例演示 下面通过实例演示方法Sleep的使用。

```
using UnityEngine;
using System.Collections;

public class SleepOrWake_ts : MonoBehaviour
{
    public Rigidbody A;
    void Awake()
    {
        A.Sleep();
    }
    void OnGUI()
    {
        if (GUI.Button(new Rect(10.0f, 10.0f, 200.0f, 45.0f), "Sleep"))
        {
            if (!A.IsSleeping())
            {
                A.Sleep();
            }
        }
        if (GUI.Button(new Rect(10.0f, 60.0f, 200.0f, 45.0f), "WakeUp"))
        {
            if (A.IsSleeping())
            {
                A.WakeUp();
            }
        }
        Debug.Log("A物体的Y坐标： " + A.transform.position.y+"     A物体是否处于休眠状态：
           "+A.IsSleeping());
    }
}
```

在这段代码中，首先声明了一个Rigidbody变量A，然后在方法Awake中将刚体A置于休眠状态，最后在方法OnGUI中定义了两个不同功能的Button，用于控制刚体A的休眠状态，并打印刚体A的状态，如图10-15所示。由打印结果可知，尽管A物体受重力感应，但当调用方法Sleep时会立即停止下落，直到调用方法WakeUp将其唤醒为止。

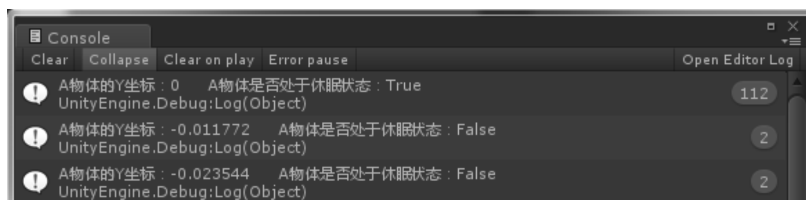


图10-15 实例演示运行结果截图

10.2.9 SweepTest方法：检测碰撞器

基本语法 (1) public bool SweepTest(Vector3 direction, out RaycastHit hitInfo);

(2) public bool SweepTest(Vector3 direction, out RaycastHit hitInfo, float distance);

其中参数direction为探测方向，参数distance为有效探测距离，默认为无穷大。

功能说明 此方法用于检测在刚体的direction方向是否有碰撞器对象，且对象的有效探测距离不大于distance。例如，设有GameObject实例A和B，如图10-16所示，它们的边长都为1，rotation都为0，A物体含有组件Rigidbody，B物体在A物体的右侧。设它们的坐标分别为A(1.0f,1.0f,1.0f)、B(4.0f,1.0f,1.0f)，则执行以下代码后：

```
A.rigidbody.SweepTest(A.transform.right,out hit,10.0f);
```

- hit.distance=2.0f，即从A物体的右表面到B物体左表面的距离，而非它们重心坐标之间的距离4–1=3.0f。
- B物体中只要含有Collider组件即可，无需含有Rigidbody组件。
- A的有效探测距离为10，所以当B.x>12时，即使B在A的右侧，A物体也无法探测B的存在，即返回值为false。
- direction的方向为A在世界坐标系中的方向。
- 若在B的右侧还有一个物体C也在A的有效探测范围内，则hit探测到B时就停止了，不会再向右继续探测，即SweepTest的返回结果只是第一个被探测到的物体。

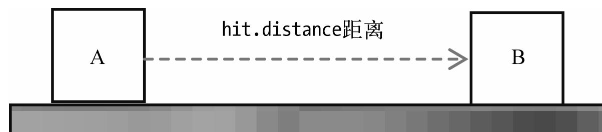


图10-16 有效探测距离示意图

实例演示 下面通过实例演示方法SweepTest的使用。

```
using UnityEngine;
using System.Collections;

public class SweepTest_ts : MonoBehaviour
{
    public GameObject A, B;
    RaycastHit hit;
    float len = 10.0f;//有效探测距离
    void Start()
    {
        A.transform.position = new Vector3(1.0f, 1.0f, 1.0f);
        B.transform.position = new Vector3(4.0f, 1.0f, 1.0f);
    }

    void FixedUpdate()
```

```
    {
        //探测刚体A右侧len距离内是否存在物体
        if (A.rigidbody.SweepTest(A.transform.right, out hit, len))
        {
            Debug.Log("A物体右侧存在物体：" + hit.transform.name + " 其距离A的距离为：" +
                hit.distance);
        }
        else
        {
            Debug.Log("A物体右侧" + len + "米范围内没检测到带碰撞器的物体！");
        }
    }

    void OnGUI()
    {
        if (GUI.Button(new Rect(10.0f, 10.0f, 200.0f, 45.0f), "设置B坐标使A无法探测到"))
        {
            //重置B的position，使得物体A、B的间距大于len值
            B.transform.position = new Vector3(12.0f, 1.0f, 1.0f);
        }
        if (GUI.Button(new Rect(10.0f, 60.0f, 200.0f, 45.0f), "取消B中的Rigidbody组件"))
        {
            //销毁B物体中的Rigidbody组件
            //运行程序可以发现，B物体中是否存在Rigidbody对探测结果没有影响
            if (B.GetComponent<Rigidbody>())
            {
                Destroy(B.GetComponent<Rigidbody>());
            }
        }
        if (GUI.Button(new Rect(10.0f, 110.0f, 200.0f, 45.0f), "取消B中的Collider组件"))
        {
            //销毁B物体中的Collider组件
            //运行程序可以发现，如果B中无Collider组件则A无论如何也探测不到B的存在
            if (B.GetComponent<Collider>())
            {
                Destroy(B.GetComponent<Collider>());
            }
        }
        //对B物体的状态重置
        if (GUI.Button(new Rect(10.0f, 160.0f, 200.0f, 45.0f), "重置"))
        {
            B.transform.position = new Vector3(4.0f, 1.0f, 1.0f);
            if (!B.GetComponent<Collider>())
            {
                B.AddComponent<BoxCollider>();
            }
            if (!B.GetComponent<Rigidbody>())
            {
                B.AddComponent<Rigidbody>();
                B.rigidbody.useGravity = false;
            }
        }
    }
}
```

在这段代码中，首先声明了两个GameObject变量A和B，用于指向场景中的物体，并在Start方法中对物体A、B的position进行重置。然后在OnGUI方法中定义了4个Button用于演示不同状况下A物体的探测结果，具体内容如代码注释所述。最后在FixedUpdate方法中调用方法SweepTest，来探测刚体A右侧len距离内是否存在物体，并将结果打印出来。请自行运行程序查看，图10-17是一张Debug输出截图。

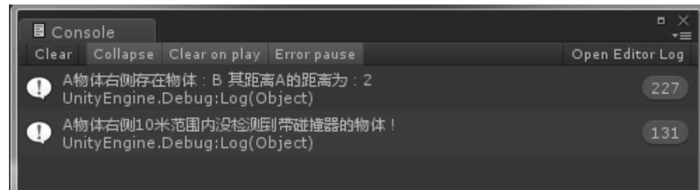


图10-17　SweepTest实例演示的运行结果

10.2.10　SweepTestAll方法：探测碰撞器

基本语法　(1) public RaycastHit[] SweepTestAll(Vector3 direction);

(2) public RaycastHit[] SweepTestAll(Vector3 direction, float distance);

其中参数direction为探测方向，参数distance为有效探测距离。

功能说明　此方法用于探测刚体的direction方向的distance距离内是否含有碰撞器，并返回所有探测到的物体的RaycastHit。此方法与方法SweepTest功能相似，可以参考其功能说明，不同的是，此方法探测的是有效范围内的所有物体，并返回每个探测到的物体的RaycastHit。

实例演示　下面通过实例演示方法SweepTestAll的使用。

```
using UnityEngine;
using System.Collections;

public class SweepTestAll_ts : MonoBehaviour
{
    public GameObject A, B, C, D;
    RaycastHit[] hits;
    float len = 10.0f;//有效探测距离
    void Start()
    {
        A.transform.position = new Vector3(1.0f, 1.0f, 1.0f);
        B.transform.position = new Vector3(4.0f, 1.0f, 1.0f);
        C.transform.position = new Vector3(7.0f, 1.0f, 1.0f);
        //D物体超出了A的有效探测距离，不会被探测到
        D.transform.position = new Vector3(12.0f, 1.0f, 1.0f);
        hits = A.rigidbody.SweepTestAll(A.transform.right, len);
        float l = hits.Length;
        Debug.Log("A探测到的物体个数为：" + l + " 它们分别是：");
        //遍历探测结果
```

```
            foreach (RaycastHit hit in hits)
            {
                Debug.Log(hit.transform.name);
            }
        }
    }
```

在这段代码中，首先声明了4个GameObject类型变量A、B、C、D，用于指向场景中物体，然后在Start方法中对这4个变量的位置进行初始化，其中在设置对象D的位置时，使其与A的距离大于刚体A的有效探测距离len。接着接着调用方法SweepTestAll探测刚体A右侧len距离内的物体，最后遍历探测结果并打印，结果如图10-18所示。

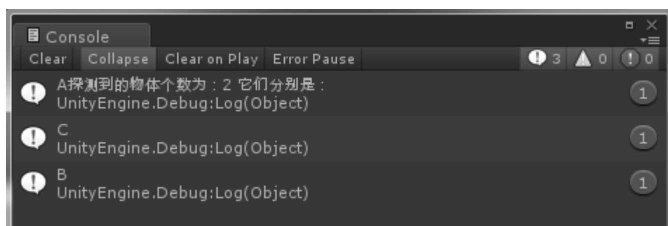


图10-18 SweepTestAll实例演示的运行结果

10.2.11 WakeUp方法：唤醒刚体

基本语法 public void WakeUp();

功能说明 此方法用于将刚体从休眠状态唤醒。要将刚体从休眠状态唤醒，除了调用WakeUp方法外，在发生以下4种情况时，刚体会被自动唤醒。

- ❑ 其他刚体与休眠中的刚体发生了碰撞；
- ❑ 使用关节连接的其他刚体发生了移动；
- ❑ 刚体的属性发生了改变；
- ❑ 给休眠中的刚体施加了一个外力。

实例演示 请参考方法Sleep的实例演示。

10.3 关于 useGravity、isKinematic 和 velocity 的使用注解

功能区别如下。

- ❑ useGravity属性用来确定刚体是否接受重力加速度的感应。
- ❑ isKinematic属性用来确定刚体是否接受动力学模拟，此影响不仅包括重力感应，还包括速度、阻力、质量等的物理模拟。

如图10-19所示，A和B为两个刚体物体，A在B的正上方，开始时A和B的重力感应都被关闭，都

10

处于静止状态，且接受动力学模拟即isKinematic为false。现在开启A的重力感应，则A从①处开始加速下落，当下落到②处时，关闭A的重力感应，但isKinematic依然为false（即接受动力学模拟），则A将以当前速度匀速下落。但是此时若关闭物理感应，即isKinematic=true，则A将立即停止移动。当A与B发生碰撞时，若B的重力感应依然关闭，但接受动力学模拟，即isKinematic=false，则根据动量守恒B将产生一个向下的速度。但是若关闭B物体的动力学模拟，即isKinematic=true，则B保持静止，不会因受到A的碰撞而下落。

在刚体不与其他物体接触的情况下，velocity的值只与Gravity、drag及Kinematic有关，与质量mass及物体的Scale值无关。

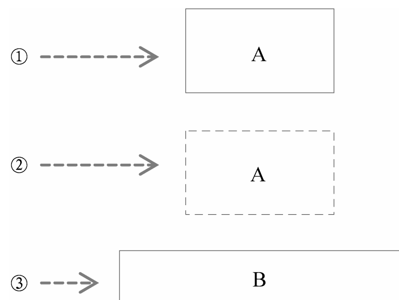


图10-19　物体下落位置示意图

实例演示　下面通过实例演示useGravity、isKinematic和velocity的使用。

```
using UnityEngine;
using System.Collections;

public class GraAndKin_ts : MonoBehaviour
{
    public Rigidbody A, B;
    string str_AG = "";
    string str_AK = "";
    string str_BK = "";
    Vector3 v1, v2;
    void Start()
    {
        //为了更好地演示，将重力加速度降低
        Physics.gravity = new Vector3(0.0f,-0.5f,0.0f);
        A.useGravity = false;
        B.useGravity = false;
        A.isKinematic = false;
        B.isKinematic = false;
        str_AG = "A开启重力感应";
        str_AK = "A关闭物理感应";
        str_BK = "B关闭物理感应";
        v1 = A.position;
        v2 = B.position;
    }
```

```
    void OnGUI()
    {
        if (GUI.Button(new Rect(10.0f, 10.0f, 200.0f, 45.0f), str_AG))
        {
            if (A.useGravity)
            {
                A.useGravity = false;
                str_AG = "A开启重力感应";
            }
            else
            {
                A.useGravity = true;
                str_AG = "A关闭重力感应";
            }
        }
        if (GUI.Button(new Rect(10.0f, 60.0f, 200.0f, 45.0f), str_AK))
        {
            if (A.isKinematic)
            {
                A.isKinematic = false;
                str_AK = "A关闭物理感应";
            }
            else
            {
                A.isKinematic = true;
                str_AK = "A开启物理感应";
            }
        }
        if (GUI.Button(new Rect(10.0f, 110.0f, 200.0f, 45.0f), str_BK))
        {
            if (B.isKinematic)
            {
                B.isKinematic = false;
                str_BK = "B关闭物理感应";
            }
            else
            {
                B.isKinematic = true;
                str_BK = "B开启物理感应";
            }
        }
        if (GUI.Button(new Rect(10.0f, 160.0f, 200.0f, 45.0f), "重置"))
        {
            A.position = v1;
            A.rotation = Quaternion.identity;
            B.position = v2;
            B.rotation = Quaternion.identity;
        }
    }
}
```

在这段代码中，首先声明了两个Rigidbody类变量A和B，用于指向场景中的物体，然后在Start方法中对A、B的useGravity、isKinematic及position进行初始化，最后在

OnGUI方法中定义了4个Button，用于演示useGravity、isKinematic和velocity的关系。程序运行情况请参考功能说明部分，请自行运行程序，查看不同状态下变量A、B所指向物体的运动状态。

10.4　关于 Rigidbody 中 mass、density 及 scale 之间的关系注解

在Rigidbody类中，mass、density和scale这3个API之间有着较为紧密的联系，下面对它们的关系进行说明。

- 若在脚本中未使用Rigidbody. SetDensity (density : float)方法设置刚体的密度，则刚体的质量mass值为在Inspector面板中Mass的大小，此时mass与Transform中的scale大小无关。
- 若在脚本中使用Rigidbody. SetDensity (density : float)方法设置了刚体的密度，则刚体的质量为mass=density*scale.x*scale.y*scale.z，而与Inspector面板中Mass的设置大小无关。
- 若在脚本中既设置了密度Density，又设置了质量mass，则刚体实际质量值要看脚本中代码执行的前后次序了。若先执行密度设置代码，再执行质量设置代码，则刚体质量便不与密度density及物体放缩值scale有关。相反，若先执行质量设置代码，再执行密度设置代码，则质量设置代码失效。
- 当两物体发生碰撞时遵守动量守恒定理，即m1*v1+m2*v2=(m1+m2)*v，其中速度v1、v2及v按运动方向分正负号。

实例演示　下面使用实例演示Rigidbody中mass、density及scale之间的作用关系。

```
using UnityEngine;
using System.Collections;

public class Mds_ts : MonoBehaviour
{
public Rigidbody A, B;
//全局静态变量，用于与A、B刚体中脚本共享
    public static bool is_AC = false, is_BC = false;
    public static Vector3 A_v = Vector3.zero, B_v = Vector3.zero;

    void Start()
    {
        //如果不使用SetDensity则scale的大小对刚体质量无影响
        Debug.Log("A的质量mass：" + A.mass);
        A.transform.localScale = A.transform.localScale * 2.0f;
        Debug.Log("scale值放大后A的质量mass：" + A.mass);
        A.transform.localScale = A.transform.localScale / 2.0f;

        //注意，如果要使用SetDensity请先设定scale值再设置Density
        //否则一旦density确定质量也就确定了，再去设置scale将对质量改变不起作用
        A.transform.localScale = A.transform.localScale * 2.0f;
        A.SetDensity(2.0f);
        Debug.Log("改变density和scale值后A的质量mass：" + A.mass);
```

```
            //给A一个初始速度
            A.velocity = new Vector3(0.0f, 0.0f, 10.0f);
            //碰撞前动量
            Debug.Log("碰撞前mA*vA+mB*vB=" + (A.mass * (A.velocity.x + A.velocity.y + A.velocity.z)
                + B.mass * (B.velocity.x + B.velocity.y + B.velocity.z)));
        }

        void FixedUpdate()
        {
            if (is_AC && is_BC)
            {
                //碰撞后动量
                Debug.Log("碰撞后mA*vA+mB*vB=" + (A.mass * (A.velocity.x + A.velocity.y +
                    A.velocity.z) + B.mass * (B.velocity.x + B.velocity.y + B.velocity.z)));
            }
        }
    }
```

在这段代码中，首先声明了两个Rigidbody类变量A和B，用于指向场景中物体，然后在Start方法中演示了SetDensity、scale和mass之间的关系，如代码注释中所述。最后在FixedUpdate方法中检验了物体间碰撞是否符合动量守恒定理，程序运行结果如图10-20所示。

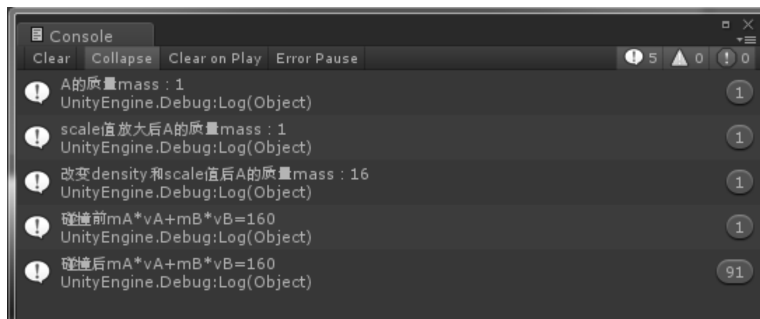


图10-20　实例演示运行结果

10.5　关于作用力方式 ForceMode 的功能注解

ForceMode为枚举类型，用来控制力的作用方式，有4个枚举成员，下面将一一介绍。其中设刚体质量均为m=2.0f，力向量均为f=(10.0f,0.0f,0.0f)。

- **ForceMode.Force**：默认方式，使用刚体的质量计算，以每帧间隔时间为单位计算动量。设FixedUpdate()的执行频率采用系统默认值（即0.02s），则由动量定理

  $$F \cdot t=m \cdot v$$

  可得10*0.02=2*v1，从而可得v1=0.1，即每帧刚体在x轴上值增加0.1米，从而可计算得刚体的每秒移动速度为v2=(1/0.02)*v1=5m/s。

- **ForceMode.Acceleration**：在此种作用方式下，会忽略刚体的实际质量而采用默认值1.0f，时间间隔以系统帧频间隔计算（默认值为0.02s），即

$$f \cdot t=1.0 \cdot v$$

可得v1= f・t=10*0.02=0.2，即刚体每帧增加0.2米，从而可得刚体的每秒移动速度为

$$v2=(1/0.02)*v1=10m/s$$

- **ForceMode.Impulse**：此种方式采用瞬间力作用方式，即把t的值默认为1，不再采用系统的帧频间隔，即

$$f \cdot 1.0=m \cdot v$$

可得v1=f/m=10.0/2.0=5.0，即刚体每帧增加5.0米，从而可得刚体每秒的速度为

$$v2=(1/0.02)*5.0=250m/s$$

- **ForceMode.VelocityChange**：此种作用方式下将忽略刚体的实际质量，采用默认质量1.0，同时也忽略系统的实际帧频间隔，采用默认间隔1.0，即

$$f \cdot 1.0=1.0 \cdot v$$

可得v1=f=10.0，即刚体每帧沿x轴移动距离为10米，从而可得刚体每秒的速度为

$$v2=(1/0.02)*v1=500m/s$$

实例演示　下面通过实例演示作用力方式ForceMode中各种作用力类型的使用。

```
using UnityEngine;
using System.Collections;

public class ForceMode_ts : MonoBehaviour
{
    public Rigidbody A, B, C, D;
    //作用力向量
    Vector3 forces = new Vector3(10.0f, 0.0f, 0.0f);

    void Start()
    {
        //初始化4个刚体的质量，使其相同
        A.mass = 2.0f;
        B.mass = 2.0f;
        C.mass = 2.0f;
        D.mass = 2.0f;
        //对A、B、C、D采用不同的作用力方式
        //注意此处只是对物体增加了1帧的作用力
        //如果要对刚体产生持续作用力请把以下代码放在FixedUpdate()方法中
        A.AddForce(forces, ForceMode.Force);
        B.AddForce(forces, ForceMode.Acceleration);
        C.AddForce(forces, ForceMode.Impulse);
        D.AddForce(forces, ForceMode.VelocityChange);
    }
```

```
    void FixedUpdate()
    {
        Debug.Log("ForceMode.Force作用方式下A每帧增加的速度：" + A.velocity);
        Debug.Log("ForceMode.Acceleration作用方式下B每帧增加的速度：" + B.velocity);
        Debug.Log("ForceMode.Impulse作用方式下C每帧增加的速度：" + C.velocity);
        Debug.Log("ForceMode.VelocityChange作用方式下D每帧增加的速度：" + D.velocity);
    }
}
```

在这段代码中，首先声明了4个Rigidbody变量和一个Vector3变量，然后在Start方法中将4个刚体的质量都设为相同的值，并分别对4个刚体施加相同的力向量，但使用不同的作用方式。最后在FixedUpdate方法中分别打印出4个刚体的速度，如图10-21所示，对于输出结果的计算方法请参考功能注解部分。

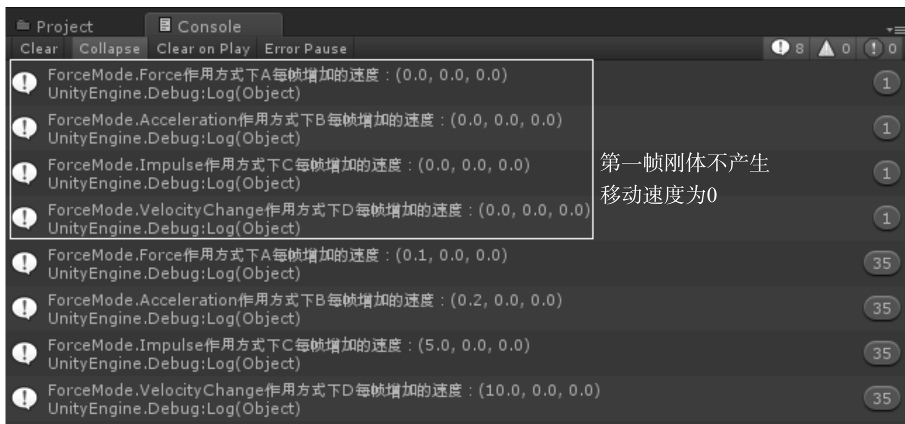


图10-21 实例演示运行结果

10.6 关于 OnTriggerXXX 和 OnCollisionXXX 的功能注解

OnTriggerXXX指的是OnTriggerEnter、OnTriggerExit和OnTriggerStay这 3 个消息，OnCollisionXXX指的是OnCollisionEnter、OnCollisionExit和OnCollisionStay这 3 个消息，它们都是用来处理不同物体在不同状态下消息的反馈，对它们的使用说明如下。

设现有A、B两个物体，且A物体正向B移动，B物体保持静止状态，如图10-22所示。

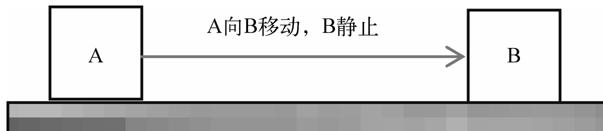


图10-22 物体位置示意图

则

- 若A中无Rigidbody组件，则B中无论是否含有Rigidbody组件，A都将穿越B物体，并且A和B脚本中的OnTriggerXXX和OnCollisionXXX方法都不会被调用。
- 若A中含有Rigidbody组件，则B中无论是否还有Rigidbody组件，只要B中含有Collider类组件，A和B脚本中的OnTriggerXXX方法或OnCollisionXXX方法就会被调用，到底调用哪一种静态方法要看A和B物体中Collider类组件中的IsTrigger是否被选中。总之，要激活OnTriggerXXX方法或OnCollisionXXX方法必须使移动的物体中含有Rigidbody组件。
- 若A中含有Rigidbody组件，B中含有Collider类组件，当A和B物体中Collider类组件的IsTrigger都没有选中时（即在Inspector面板中IsTrigger不要打勾），A和B脚本中OnCollisionXXX类的方法就会被调用，而OnTriggerXXX静态方法则不会被调用。
- 若A中含有Rigidbody组件，B中含有Collider类组件，当A和B物体中的Collider类组件的IsTrigger至少有一个被选中时，A和B物体脚本中的OnTriggerXXX静态方法会被调用，而OnCollisionXXX静态方法不会被调用。
- 当符合OnCollisionXXX静态方法激活条件时，A不可穿越B物体，A会与B发生弹性碰撞。
- 当符合OnTriggerXXX静态方法激活条件时，A会穿越B物体，即A、B物体的运动行为互不影响，只是反馈了两个物体的接触状态：未接触、开始接触、接触中、互相分离。
- OnTriggerEnter或OnCollisionEnter方法会在A刚开始接触B时被调用，且在A、B分离前只被调用一次。
- OnTriggerStay或OnCollisionStay方法会在A和B保持接触状态时被调用，且在A、B分离前每帧都会被调用。
- OnTriggerExit或OnCollisionExit方法会在A、B刚分离时被调用，且只被调用一次。

实例演示　下面通过实例演示OnTriggerXXX类方法和OnCollisionXXX类方法的使用，在本实例演示中，包括主程序脚本和物体A、B中的脚本。

- 主程序脚本，用于控制A、B的移动。

```
using UnityEngine;
using System.Collections;

public class TriggerOrCollision_ts : MonoBehaviour
{
    public GameObject A, B;
    Vector3 p_a, p_b;
    int which_change = -1;
    //将物体A、B的初始位置赋给p_a和p_b，用于重置物体组件时使用
    void Start()
    {
        p_a = A.transform.position;
        p_b = B.transform.position;
```

```
}
//控制物体A的移动
void FixedUpdate()
{
    if (which_change == 0)
    {
        A.transform.Translate(Vector3.forward * Time.deltaTime);
    }
}
void OnGUI()
{
    //当A物体无Rigidbody组件时
    //无论B是否有Rigidbody都不会激活A和B脚本中的OnCollisionXXX或OnTriggerXXX方法
    if (GUI.Button(new Rect(10.0f, 10.0f, 280.0f, 45.0f), "A物体无Rigidbody组件"))
    {
        inists();
        which_change = 0;
        if (A.GetComponent<Rigidbody>())
        {
            Destroy(A.GetComponent<Rigidbody>());
        }
    }
    //当A物体有Rigidbody组件时
    //一定会激活A和B脚本中的OnCollisionXXX或OnTriggerXXX方法
    if (GUI.Button(new Rect(10.0f, 60.0f, 280.0f, 45.0f), "A有Rigidbody组件，
    B无Rigidbody
          组件"))
    {
        inists();
        which_change = 1;
        if (!A.GetComponent<Rigidbody>())
        {
            A.AddComponent<Rigidbody>();
            A.rigidbody.useGravity = false;
        }
        if (B.GetComponent<Rigidbody>())
        {
            Destroy(B.GetComponent<Rigidbody>());
        }
        A.rigidbody.velocity = Vector3.forward;
    }
    //当A物体有Rigidbody组件时
    //且A与B物体IsTrigger都未选中时，只会激活A和B脚本中的OnCollisionXXX方法
    if (GUI.Button(new Rect(10.0f, 110.0f, 280.0f, 45.0f), "A与B物体IsTrigger
    都未选中"))
    {
        inists();
        which_change = 2;
        A.GetComponent<Collider>().isTrigger = false;
        B.GetComponent<Collider>().isTrigger = false;
        if (!A.GetComponent<Rigidbody>())
        {
            A.AddComponent<Rigidbody>();
            A.rigidbody.useGravity = false;
```

10

```
                }
                A.rigidbody.velocity = Vector3.forward;
            }
            //当A物体有Rigidbody组件时
            //且A与B物体IsTrigger至少有一个被选中时，只会激活A和B脚本中的OnTriggerXXX方法
            if (GUI.Button(new Rect(10.0f, 160.0f, 280.0f, 45.0f), "A物体IsTrigger被选中"))
            {
                inists();
                which_change = 3;
                A.GetComponent<Collider>().isTrigger = true;
                if (!A.GetComponent<Rigidbody>())
                {
                    A.AddComponent<Rigidbody>();
                    A.rigidbody.useGravity = false;
                }
                A.rigidbody.velocity = Vector3.forward;
            }
            if (GUI.Button(new Rect(10.0f, 210.0f, 280.0f, 45.0f), "重置"))
            {
                inists();
                which_change = 4;
            }
        }
        //初始化数据
        void inists()
        {
            if (A.GetComponent<Rigidbody>())
            {
                A.rigidbody.velocity = Vector3.zero;
                A.rigidbody.angularVelocity = Vector3.zero;
            }
            if (B.GetComponent<Rigidbody>())
            {
                B.rigidbody.velocity = Vector3.zero;
                B.rigidbody.angularVelocity = Vector3.zero;
            }
            A.transform.position = p_a;
            A.transform.rotation = Quaternion.identity;
            B.transform.position = p_b;
            B.transform.rotation = Quaternion.identity;
        }
    }
```

在这段代码中，首先声明了两个GameObject类型的变量A和B，然后在Start方法中将A、B物体的初始位置赋给p_a和p_b，用于重置物体时使用，最后在OnGUI方法中定义了 5 个不同功能的Button，用于演示OnTriggerXXX类方法和OnCollisionXXX类方法的不同作用，具体请参考代码中的注释。

- A物体的脚本（B物体的脚本与此类似，不再赘述。）

```
using UnityEngine;
using System.Collections;
```

```
public class ATorC_ts : MonoBehaviour
{
    //开始接触
    void OnTriggerEnter(Collider other)
    {
        Debug.Log("A物体的OnTriggerEnter被调用，被接触的物体为" + other.name);
    }
    //结束接触
    void OnTriggerExit(Collider other)
    {
        Debug.Log("A物体的OnTriggerExit被调用，被接触的物体为" + other.name);
    }
    //保持接触
    void OnTriggerStay(Collider other)
    {
        Debug.Log("A物体的OnTriggerStay被调用，被接触的物体为" + other.name);
    }
    //开始碰撞
    void OnCollisionEnter(Collision collision)
    {
        Debug.Log("A物体的OnCollisionEnter被调用，被碰撞的物体为" +
            collision.gameObject.name);
    }
    //退出碰撞
    void OnCollisionExit(Collision collision)
    {
        Debug.Log("A物体的OnCollisionExit被调用，被碰撞的物体为" +
            collision.gameObject.name);
    }
    //保持碰撞
    void OnCollisionStay(Collision collision)
    {
        Debug.Log("A物体的OnCollisionStay被调用，被碰撞的物体为" +
            collision.gameObject.name);
    }
}
```

在这段代码中，只是在OnTriggerXXX类方法和OnCollisionXXX类方法中打印出A物体被接触或被碰撞后的相应信息，请自行运行程序，观察在不同操作条件下的Debug输出结果。

第11章 Time类

11

Time类是Unity中获取时间信息的接口类，只有静态属性。本章简要介绍了Time类的一些静态属性。

11.1 Time 类静态属性

在Time类中，涉及的静态属性有realtimeSinceStartup、smoothDeltaTime和time属性，在介绍time属性时涉及了Time类的多个其他属性的使用，具体请查看time属性的实例演示中的相关介绍。下面详细介绍这些属性。

11.1.1 realtimeSinceStartup属性：程序运行实时时间

基本语法 public static float realtimeSinceStartup { get; }

功能说明 此属性用于返回从游戏启动到现在已运行的实时时间（只读），以秒为单位。此属性通常可用Time.time代替使用，但realtimeSinceStartup的返回值不受timeScale属性变化的影响。

实例演示 下面通过实例演示属性realtimeSinceStartup的使用。

```
using UnityEngine;
using System.Collections;

public class RealtimeSinceStartup_ts : MonoBehaviour
{
    public Rigidbody rg;
    void Start()
    {
        Debug.Log("Time.timeScale的默认时间：" + Time.timeScale);
        //观察刚体在timeScale变化前后的移动速度
        rg.velocity = Vector3.forward * 2.0f;
        Time.timeScale = 0.5f;
    }
    void Update()
    {
        Debug.Log("Time.timeScale的当前值：" + Time.timeScale);
        Debug.Log("Time.time:" + Time.time);
        Debug.Log("Time.realtimeSinceStartup:" + Time.realtimeSinceStartup);
```

```
    }
    void OnGUI()
    {
        if (GUI.Button(new Rect(10.0f, 10.0f, 200.0f, 45.0f), "Time.timeScale=0.5f"))
        {
            Time.timeScale = 0.5f;
        }
        if (GUI.Button(new Rect(10.0f, 60.0f, 200.0f, 45.0f), "Time.timeScale=1.0f"))
        {
            Time.timeScale = 1.0f;
        }
    }
}
```

在这段代码中，首先声明了一个Rigidbody变量rg，并在Start方法中给刚体rg一个初始速度，然后在方法OnGUI中定义了两个Button用来控制Time.timeScale的值，最后在Update方法中分别打印出了Time.timeScale、Time.time和Time.realtimeSinceStartup的值，图11-1为一张运行时截图。

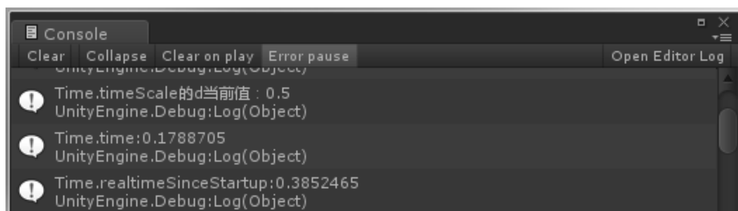


图11-1　realtimeSinceStartup实例演示的运行结果

11.1.2　smoothDeltaTime属性：平滑时间间隔

基本语法　public static float smoothDeltaTime { get; }

功能说明　此属性用于返回Time.deltaTime的平滑输出值（只读）。Time.smoothDeltaTime比Time.deltaTime的波幅震荡更平滑（如图11-2和图11-3所示），通常Time.smoothDeltaTime的累加和比Time.deltaTime的累加和稍微大些。Time.smoothDeltaTime主要用于在非FixedUpdate方法中需要平滑过渡的计算。

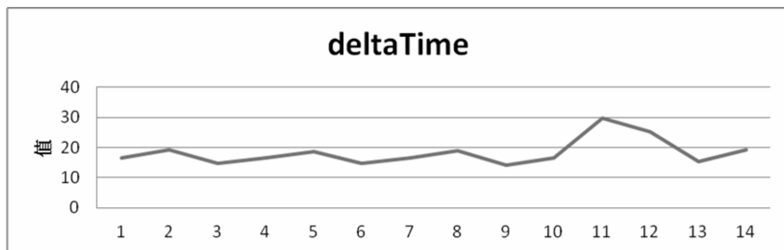


图11-2　deltaTime示意图

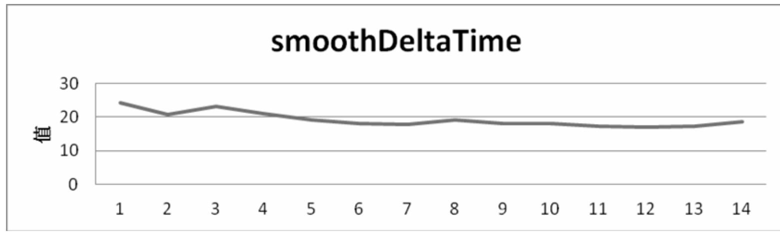


图11-3　smoothDeltaTime示意图

实例演示　下面通过实例演示属性smoothDeltaTime的使用。

```
using UnityEngine;
using System.Collections;

public class SmoothDeltaTime_ts : MonoBehaviour
{
    float a = 0, b = 0;
    void Update()
    {
        float t1, t2;
        t1 = Time.deltaTime;
        t2 = Time.smoothDeltaTime;
        Debug.Log("Time.deltaTime:" + t1);
        Debug.Log("smoothDeltaTime:" + t2);
        a += t1;
        b += t2;
        Debug.Log("Time.deltaTime的累加和:" + a + " smoothDeltaTime的累加和:" + b);
    }
}
```

在这段代码中，首先声明了两个变量a和b，然后在Update方法中将Time.deltaTime和Time.smoothDeltaTime的值赋给t1和t2，并将t1、t2的值累加到变量a、b中，最后分别打印出t1、t2、a、b这4个变量的值，如图11-4所示。

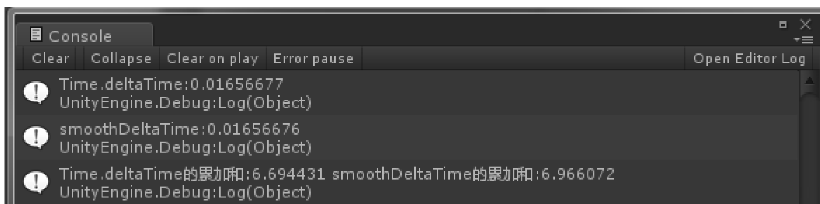


图11-4　实例演示运行结果截图

11.1.3　time属性：程序运行时间

基本语法　`public static float time { get; }`

功能说明　此属性用于返回从游戏启动到现在的运行时间（只读），以秒为单位。

实例演示 下面通过实例演示Time类一些属性的使用。

```
using UnityEngine;
using System.Collections;

public class Time_ts : MonoBehaviour
{
    float t1 = 0.0f;
    void Update()
    {
        // deltaTime属性用于返回从上一帧到现在所经历的时间（只读），以秒为单位
        //由于Update()函数中的代码是以帧来执行的，其执行的时间间隔是不固定的
        //当需要以秒为单位对某个物体进行变换时就需要和Time.deltaTime结合使用
        t1 = Time.deltaTime;
    }
    void OnGUI()
    {
        if (GUI.Button(new Rect(10.0f, 10.0f, 200.0f, 45.0f), "加载新场景"))
        {
            Application.LoadLevel("newScene01_unity");
        }
        GUI.Label(new Rect(10.0f, 60.0f, 300.0f, 45.0f), "当前游戏场景名字:" + Application.
            loadedLevelName);
        GUI.Label(new Rect(10.0f, 110.0f, 300.0f, 45.0f), "当前游戏已运行时间Time.time:" +
            Time.time);
        //属性timeSinceLevelLoad用于返回从最后关卡加载完成到现在所经历的时间（只读），以秒为
            单位
        GUI.Label(new Rect(10.0f, 160.0f, 400.0f, 45.0f), "当前场景已运行时间
            Time.timeSinceLevelLoad：" + Time.timeSinceLevelLoad);
        GUI.Label(new Rect(10.0f, 210.0f, 300.0f, 45.0f), "上一帧消耗的时间Time.deltaTime:"
            + t1);
        //属性fixedTime用于返回从游戏启动到现在以固定频率更新的时间（只读）
        GUI.Label(new Rect(10.0f, 260.0f, 300.0f, 45.0f), "固定增量时间Time.fixedTime:"
            + Time.fixedTime);
        //属性fixedDeltaTime用于返回以固定频率更新时，相邻两帧的时间间隔（只读）
        //fixedDeltaTime的值可以通过导航菜单栏“Edit”→“ProjectSettings”→“Time”菜单项
            中的 “FixedTimestep”进行设置
        GUI.Label(new Rect(10.0f, 310.0f, 400.0f, 45.0f), "上一帧消耗的固定增量时间
            Time.fixedDeltaTime:" + Time.fixedDeltaTime);
        //属性frameCount用于返回从游戏启动到现在已经更新的频率总数（只读）
        GUI.Label(new Rect(10.0f, 360.0f, 300.0f, 45.0f), "游戏已运行帧数Time.frameCount:" +
            Time.frameCount);
    }
}
```

在这段代码中，首先声明了一个浮点型变量t1，并在Update方法中将Time.deltaTime的返回值赋给t1，然后在OnGUI方法中使用GUI.Label方法在游戏界面中显示出了很多有关Time类的属性值，例如time属性、timeSinceLevelLoad属性、Time.frameCount属性等，具体请查看程序代码。请自行运行程序，查看场景加载前后各类数据的变化，图11-5是一张运行时截图。



图11-5　time实例演示的运行结果

11.2　Time 类其他常用静态属性功能简介

本小节简要介绍一些Time类的其他常用静态属性的功能，列举如下。

- captureFramerate属性：用于设置或返回帧速率的值。当captureFramerate的值比0大时，时间会以每帧（1.0 / captureFramerate）秒前进，不考虑真实时间，例如Time.captureFramerate = 25，则游戏的帧速率为25帧/秒。
- deltaTime属性：用于返回从上一帧到现在所经历的时间（只读），以秒为单位。
- fixedDeltaTime属性：用于返回以固定频率更新时，相邻两帧的时间间隔（只读）。
- fixedTime属性：用于返回从游戏启动到现在以固定频率更新的时间（只读）。
- frameCount属性：用于返回从游戏启动到现在已经更新的频率总数（只读）。
- maximumDeltaTime属性：用于设置或返回每帧更新可以消耗的最大时间，以秒为单位。可以通过导航菜单栏“Edit”→“ProjectSettings”→“Time”菜单项中的“MaximumAllowedTimestep”进行设置。当帧更新消耗的时间大于maximumDeltaTime时，物理和其他固定帧速率的更新将被降低，游戏运行会进入一种“迟缓”状态。
- timeScale属性：用于控制游戏时间的流逝速度，默认值为1。通常可用使用此属性来控制游戏的运行状态，例如暂停、快进等。
- timeSinceLevelLoad属性：用于返回从最后关卡加载完成到现在所经历的时间（只读），以秒为单位。

第 12 章

Transform类

12

Transform类继承自Component类，并实现了IEnumberable接口。Transform是GameObject必须拥有的一个组件，用来管理所在GameObject对象的坐标位置、旋转角度和大小缩放。由于Transform实现了IEnumberable接口，于是可以在程序中使用foreach()方法快速遍历子物体的Transform结构。我们知道，在程序执行时，foreach()方法要比for(;;)方法快，所以在无需记录遍历位置的情况下尽量使用foreach(),如以下代码所示：

```
using UnityEngine;
using System.Collections;
//将所有子物体放大2*2*2=8倍，不包括自身。
public class Foreach_ts : MonoBehaviour {
    void Start () {
        foreach (Transform child in transform)
        {
            child.localScale *= 2.0f;
        }
    }
}
```

本章主要介绍了Transform类的一些实例属性和实例方法，最后分别对localScale和lossyScale功能以及一些与坐标系转换相关的API进行了注解。

12.1 Transform 类实例属性

在Transform类中，涉及的实例属性有eulerAngles、forward、hasChanged、localPosition、localToWorldMatrix、parent和worldToLocalMatrix属性，下面详细介绍这些属性。

12.1.1 eulerAngles属性：欧拉角

基本语法 public Vector3 eulerAngles { get; set; }

功能说明 此属性用于返回或设置GameObject对象的欧拉角，对其使用说明如下。

- 在Unity3D引擎中使用四元数Quaternion来存储和表示GameObject的旋转角度，无论是在Inspector面板中对Rotation设置了怎样的数值，还是在脚本中对transform.

eulerAngles赋予了怎样的数值，程序在编译运行时都会把它们转换成Quaternion类型再计算。

- 只能对transform.eulerAngles进行整体赋值，如transform.eulerAngles=new Vector(1.0f,2.0f,3.0f)，不可以对transform.eulerAngles的单独分量（如transform.eulerAngles.x）进行赋值。
- transform.eulerAngles.x 返回值的范围为[0,90]和[270,360);transform.eulerAngles.y和transform.eulerAngles.z返回值的范围为[0 ,360)。
- 对transform.eulerAngles进行赋值或获取transform.eulerAngles的值都是相对世界坐标系而言的，若要相对transform的父物体（如果有的话）进行角度的变换则需要使用属性localEulerAngles来设置。
- 设在脚本中有代码：transform.eulerAngles=new Vector3(10.0f,20.0f,30.0f)，则GameObject对象会先沿着z轴旋转30度，再沿着x轴旋转10度，最后再沿着y轴旋转20度。注意不同的旋转执行顺序，物体的最终状态是不同的。

12.1.2　forward属性：z轴单位向量

基本语法　public Vector3 forward { get; set; }

功能说明　此属性用于返回或设置transform自身坐标系中z轴方向的单位向量对应的世界坐标系中的单位向量。transform.forward即为transform.TransformDirection(new Vector3(0.0f, 0.0f, 1.0f))的简化方式。

实例演示　下面通过实例演示Transform的right、up和forward属性的使用。

```
using UnityEngine;
using System.Collections;

public class Forward_ts : MonoBehaviour {
    void Start () {
        //先给transform一个任意的欧拉角，使得transform的局部坐标系和世界坐标系方向不一致
        transform.eulerAngles = new Vector3(15.0f,30.0f,60.0f);
        //right
        Debug.Log("right:"+transform.right);
        Debug.Log("Dir:" + transform.TransformDirection(new Vector3(1.0f, 0.0f, 0.0f)));
        //up
        Debug.Log("up:" + transform.up);
        Debug.Log("Dir:" + transform.TransformDirection(new Vector3(0.0f, 1.0f, 0.0f)));
        //forward
        Debug.Log("forward:" + transform.forward);
        Debug.Log("Dir:" + transform.TransformDirection(new Vector3(0.0f, 0.0f, 1.0f)));
    }
}
```

在这段代码的Start方法中，首先给transform一个任意的欧拉角，使得transform的局部坐标系和世界坐标系方向不一致，然后分别打印出transform的right、up和forward值，并调用TransformDirection方法实现相同的功能，结果如图12-1所示。

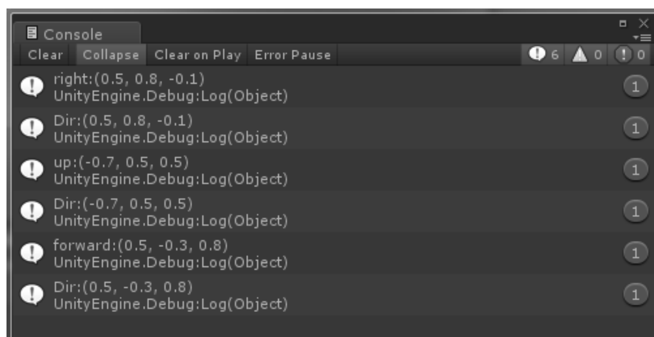


图12-1　实例演示的运行结果

12.1.3　hasChanged属性：transform组件是否被修改

基本语法　public bool hasChanged { get; set; }

功能说明　此属性用于判断GameObject对象从上次将此属性设为false以来，其transform组件的属性是否被修改过，例如当transform的position、rotation或scale属性被修改时transform.hasChanged将返回true。

提　　示　即使transform某个属性修改后的值与修改前的值相同，hasChanged的返回值仍然为true，请参考实例演示中的代码。

实例演示　下面通过实例演示属性hasChanged的使用方法。

```
using UnityEngine;
using System.Collections;

public class HasChanged_ts : MonoBehaviour
{
    bool is_rotate = false;//物体旋转控制
    // Update is called once per frame
    void Update()
    {
        //当transform.hasChanged为true时打印相关信息
        if (transform.hasChanged)
        {
            Debug.Log("The transform has changed!");
            transform.hasChanged = false;
        }
        if (is_rotate)
        {
            transform.Rotate(Vector3.up * 15.0f * Time.deltaTime);
        }
    }
    void OnGUI()
    {
```

```
            //使物体开始旋转，物体rotation被修改，hasChanged返回值为true
            if (GUI.Button(new Rect(10.0f, 10.0f, 80.0f, 45.0f), "Rotate start"))
            {
                is_rotate = true;
            }
            //停止旋转，hasChanged返回值为false
            if (GUI.Button(new Rect(10.0f, 65.0f, 80.0f, 45.0f), "Rotate stop"))
            {
                is_rotate = false;
            }
            //将transform的当前位置赋给transform.position，即transform的位置没有改变
            //但是transform的position属性已经被修改，故transform.hasChanged返回值为true
            if (GUI.Button(new Rect(10.0f, 125.0f, 80.0f, 45.0f), "use position"))
            {
                is_rotate = false;
                transform.position = transform.position;
            }
        }
    }
```

在这段代码中，首先声明了一个bool类型变量is_rotate，用于标示物体是否发生旋转，然后在OnGUI方法中定义了 3 个Button，分别用于改变物体的rotation和position。其中在修改transform的position时，position的值并没有改变，但是transform.hasChanged属性依然认为transform的position属性被重新赋值，从而返回true。请自行运行程序，查看在不同操作下程序的输出变化。

12.1.4 localPosition属性：局部坐标系位置

基本语法 public Vector3 localPosition { get; set; }

功能说明 此属性用于设置或返回GameObject对象在局部坐标系中的位置，若无父级对象则和属性Transform.position返回值相同。transform.localPosition的值受父级对象lossyScale的影响，当transform.localPosition的值增加1时，transform.position值的增量不一定是1，而是在相对父级坐标系中增加了父级的lossyScale倍大小的值。例如物体cube2的父级是cube1，cube1的父级是cube0，cube1的localScale为Vector3(c1x,c1y,c1z)，cube0的localScale为Vector3(c0x,c0y,c0z)，假设cube0和cube1都没有发生旋转，则GameObject对象在世界坐标系中的位置为：

transform.position.x=cube0.localPosition.x+cube1.localPosition.x*c0x+transform.localPosition.x*c1x*c0x;

transform.position.y和transform.position.z与此相同。

当物体cube2的局部坐标值localPosition增加Vector3(tx, ty, tz)时，其世界坐标值position的增加值分别为：

```
_x=tx*c1x*c0x;
_y=ty*c1y*c0y;
```

```
_z=tz*c1z*c0z;
```

当然，当某个父物体发生旋转时，其所有子物体都将受到影响。

实例演示 下面通过实例演示属性localPosition的使用，在本实例演示中首先在场景中创建了3个立方体对象，它们的层级关系如图12-2所示，下述代码为物体Cube2中的脚本。

```
using UnityEngine;
using System.Collections;

public class LocalPosition_ts : MonoBehaviour {
    public Transform cube0, cube1;
        void Start () {
            Debug.Log("cube0 local position:" + cube0.localPosition);
            Debug.Log("cube0 local scale:" + cube0.localScale);
            Debug.Log("cube1 local position:"+cube1.localPosition);
            Debug.Log("cube1 local scale:" + cube1.localScale);
            Debug.Log("cube2 local position:" + transform.localPosition);
            Debug.Log("cube2 world position:" + transform.position);
        }
}
```

在这段代码中，首先声明了两个Transform变量cube0和cube1，用于指向场景中的物体Cube0和Cube1，然后在Start方法中分别打印出了物体Cube0和Cube1的local position和local scale值，以及物体Cube2的local position和world position值，如图12-3所示。对输出结果的计算方法请参考功能说明部分。

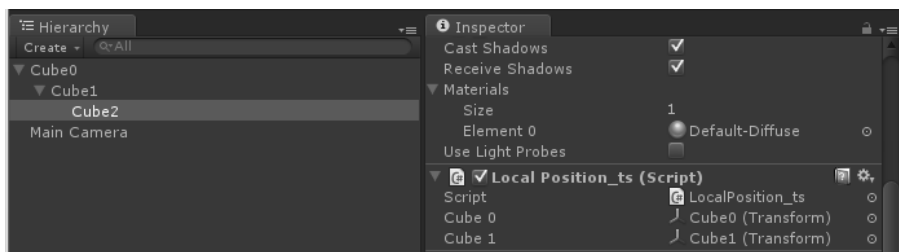


图12-2 场景物体间层级关系

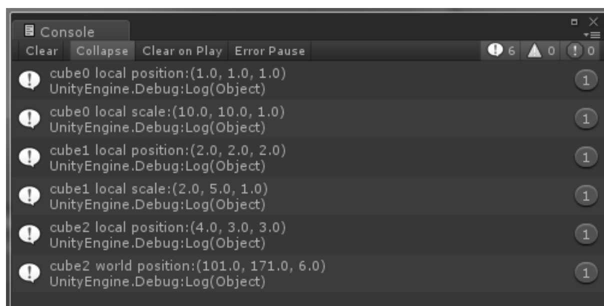


图12-3 localPosition实例演示的运行结果

12.1.5　localToWorldMatrix属性：转换矩阵

基本语法　public Matrix4x4 localToWorldMatrix { get; }

功能说明　此属性用于返回从transform局部坐标系向世界坐标系转换的Matrix4x4矩阵。例如，设A为一个Vector3，B为Transform实例，且B的x、y、z轴方向和世界坐标系方向一致，有以下代码：

```
Vector3 v1 = B.localToWorldMatrix.MultiplyPoint3x4(A);
```

则分量值v1.x=B.Position.x+A.x*B.lossyScale.x，v1.y、v1.z的值，以此类推。通俗地讲，v1的值即为A在世界坐标系中的向量。当B的x、y、z轴方向和世界坐标系方向不一致，即当B的Rotation不为0时，变换关系会更复杂，具体请参考12.4节。

提　　示　一般情况下可以用方法TransformPoint (position : Vector3)来实现Vector3实例从transform局部坐标系向世界坐标系的转换。

实例演示　下面通过实例演示属性localToWorldMatrix的使用。

```
using UnityEngine;
using System.Collections;

public class LocalToWorldMatrix_ts : MonoBehaviour {
    void Start()
    {
        //local_vector3为transform局部坐标系中的一个向量
        Vector3 local_vector3 = new Vector3(1.0f, 2.0f, 3.0f);
        Vector3 v1 = transform.localToWorldMatrix.MultiplyPoint3x4(local_vector3);
        Debug.Log("transform position:" + transform.position);
        Debug.Log("transform lossyScale:" + transform.lossyScale);
        Debug.Log("local_vector3:" + local_vector3);
        Debug.Log("local_vector3在世界坐标系中的向量为：" + v1);
    }
}
```

在这段代码的Start方法中，首先声明了一个Vector3变量local_vector3，用于表示transform局部坐标系中的一个向量，然后调用transform的localToWorldMatrix属性，并调用方法MultiplyPoint3x4求local_vector3在世界坐标系中的向量。最后打印出相关信息，打印结果如图12-4所示。对输出结果的计算方法请参考功能说明部分。

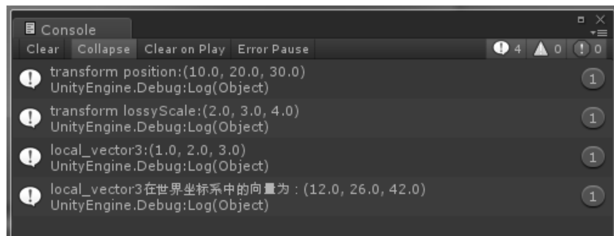


图12-4　localToWorldMatrix实例演示的运行结果

12.1.6 parent属性：父物体Transform实例

基本语法 public Transform parent { get; set; }

功能说明 此属性用于返回父物体的Transform实例。transform.parent只能返回父一级对象的Transform，若要返回父物体的父物体，可以使用transform.parent.parent，更多级父物体以此类推。若父物体不存在，则返回null。若想返回transform的最顶层的父物体，可以使用transform.root属性。

实例演示 下面通过实例演示属性parent的使用，在本实例演示中，首先在场景中创建了两个Cube物体和一个Sphere物体，它们的层级关系如图12-5所示。下面是场景中物体Sphere中的脚本。

```
using UnityEngine;
using System.Collections;

public class Parent_ts : MonoBehaviour {
    void Start () {
        //返回transform的父物体的Transform
        Debug.Log("一级父物体名字:" + transform.parent);
        //返回transform的二级父物体的Transform
        Debug.Log("二级父物体名字:" + transform.parent.parent);
        //返回transform的三级父物体的Transform
        Debug.Log("三级父物体名字:" + transform.parent.parent.parent);
        //返回transform的最顶层（root）父物体的Transform
        Debug.Log("最顶层父物体名字:" + transform.root);
  }
}
```

在这段代码的Start方法中，分别演示了transform的parent属性和root属性的使用，具体说明请参考代码注释，打印结果如图12-6所示。

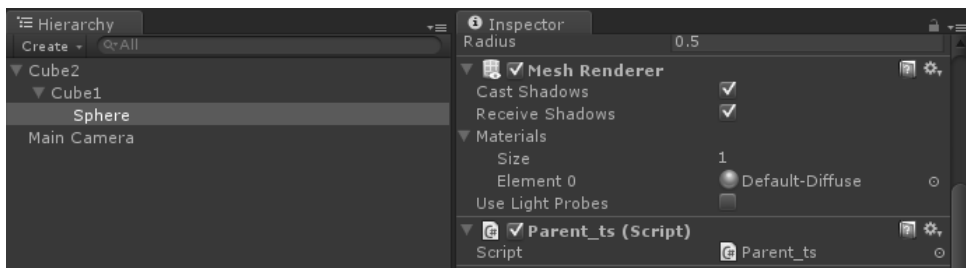


图12-5 场景中物体间层级关系

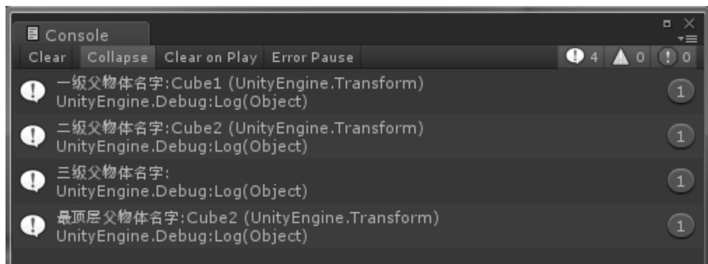


图12-6 parent实例演示的运行结果

12.1.7 worldToLocalMatrix属性：转换矩阵

基本语法 public Matrix4x4 worldToLocalMatrix { get; }

功能说明 此属性用于返回物体从世界坐标系向transform自身坐标系转换的Matrix4x4矩阵。例如，设A、B为Transform实例，且A的x、y、z轴方向和世界坐标系方向一致，有以下代码：

```
Vector3 v1 = A.worldToLocalMatrix.MultiplyPoint3x4(B.position);
```

则分量值v1.x=(B.position.x/A.lossyScale.x)-A.position.x，v1.y、v1.z的值以此类推。通俗地讲，v1的值即为B在A的坐标系中的相对位置。当A的x、y、z轴方向和世界坐标系方向不一致，即当A的Rotation不为0时，变换关系会更复杂，具体请参考12.4节。

提 示 一般情况下可以用方法InverseTransformPoint (position : Vector3)来实现Vector3实例从世界坐标系向transform自身坐标系转换。

实例演示 下面通过实例演示属性worldToLocalMatrix的使用。

```
using UnityEngine;
using System.Collections;
public class WorldToLocalMatrix_ts : MonoBehaviour {
    public Transform t1;
    void Start()
    {
        //返回t1 相对transform的向量
        Vector3 v1 = transform.worldToLocalMatrix.MultiplyPoint3x4(t1.position);
        Debug.Log("transform position:"+transform.position);
        Debug.Log("transform lossyScale:" + transform.lossyScale);
        Debug.Log("t1 position:"+t1.position);
        Debug.Log("t1 相对transform的向量为： "+v1);
    }
}
```

在这段代码中，首先声明了一个Transform变量t1，用于指向场景中的物体，然后在Start方法中调用transform的worldToLocalMatrix属性求解转换矩阵，并调用方法

MultiplyPoint3x4求t1的坐标相对transform的向量。最后打印出相关信息，打印结果如图12-7所示。

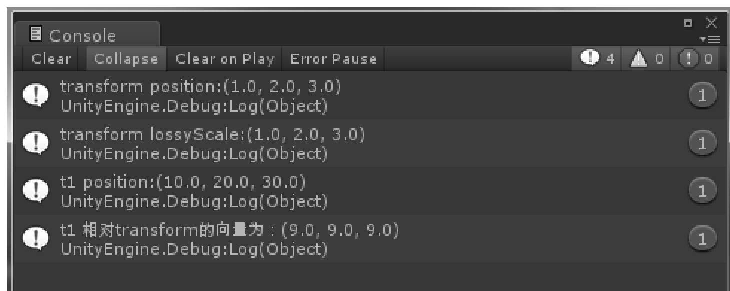


图12-7　worldToLocalMatrix实例演示的运行结果

12.2　Transform 类实例方法

在Transform类中，涉及的实例方法有DetachChildren方法、GetChild方法、InverseTransformDirection 方法、InverseTransformPoint方法、IsChildOf方法、LookAt方法、Rotate 方法、RotateAround 方法、TransformDirection方法、TransformPoint方法和Translate 方法。由于Rotate方法和Translate方法是两个非常重要且常用的方法，并且重载方法较多，于是在介绍这两个方法时按它们重载方法的不同分两次来介绍。下面详细介绍这些方法。

12.2.1　DetachChildren方法：分离物体层级关系

基本语法　public void DetachChildren();

功能说明　此方法的功能是使GameObject对象的所有子物体和自身分离层级关系，当子物体的行为不再依赖父物体时可用此方法使父子关系分离。若子物体仍有子物体，分离后的子物体将保留子物体与子物体的子物体的层级关系。如下图所示，图12-8是分离前的层级关系，图12-9是分离后的层级关系。

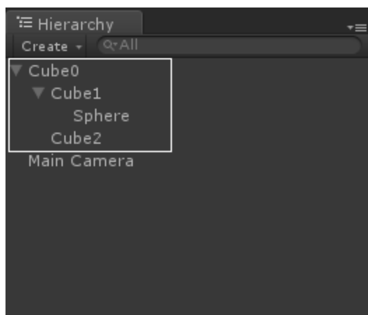


图12-8　分离前的层级关系

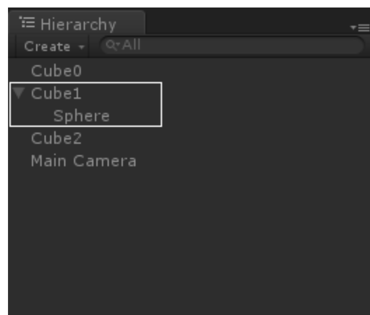


图12-9　分离后的层级关系

实例演示　下面通过实例演示方法DetachChildren的使用。在本实例演示中，首先需要在场景中创建 3 个Cube对象和一个Sphere对象，它们的层级关系如图12-8所示，然后将以下脚本附加到Cube0物体上。

```
using UnityEngine;
using System.Collections;
public class DetachChildren_ts : MonoBehaviour
{
    void Start () {
        transform.DetachChildren();
    }
}
```

在这段代码的Start方法中调用DetachChildren方法，从而将GameObject对象与其子类对象解除父子层级关系，程序运行前后的情况请参考图12-8和图12-9。

12.2.2　GetChild方法：获取GameObject对象子类

基本语法　public Transform GetChild(int index);

其中参数index为子物体索引值。

功能说明　此方法用于返回transform的索引值为index的子类Transform实例。参数index的值要小于transform的childCount值。

实例演示　下面通过实例演示方法GetChild的使用。在本实例演示中，首先在场景中创建了4个Cube对象，它们的层次关系如图12-10所示，并将下面脚本赋给物体First。

```
using UnityEngine;
using System.Collections;

public class GetChild_ts : MonoBehaviour
{
    void Start()
    {
        //transform的子物体数量
        int ct = transform.childCount;
        Debug.Log("子物体数量：" + ct);
        for (int i = 0; i < ct; i++)
        {
            Debug.Log("索引为" + i + "的子物体名字：" + transform.GetChild(i).name);
        }
    }
}
```

在这段代码的Start方法中，首先调用childCount属性，将transform的子物体数量赋给变量ct，然后调用方法GetChild遍历transform的所有子物体，并打印出子物体的名字，程序运行结果如图12-11所示。

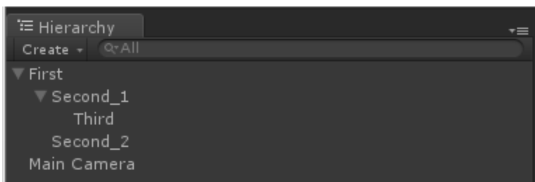


图12-10　实例演示场景中物体的层次关系

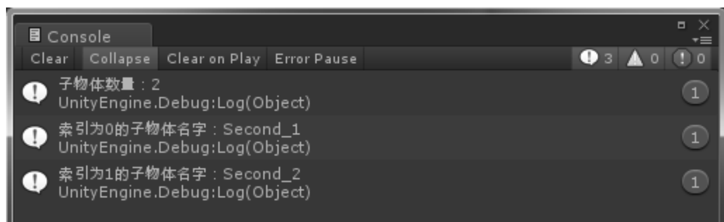


图12-11　实例演示输出结果

12.2.3　InverseTransformDirection 方法：坐标系转换

基本语法　(1) public Vector3 InverseTransformDirection(Vector3 direction);

其中参数direction为待转换的向量。

(2) public Vector3 InverseTransformDirection(float x, float y, float z);

功能说明　此方法用于将参数direction从世界坐标系转换到GameObject对象的局部坐标系。例如，设world_v为Vector3实例，且transform的x、y、z轴方向和世界坐标系的x、y、z轴方向保持一致，有以下代码：

```
Vector3 local_v = transform.InverseTransformDirection(world_v);
```

则local_v.x=world_v.x，local_v.y=world_v.y，local_v.z=world_v.z，转换后的向量local_v只与transform的Rotation和向量world_v有关，而与transform的position和scale值无关，此为与InverseTransformPoint (position : Vector3) 不同的地方。当transform的x、y、z轴方向和世界坐标系方向不一致，即当transform的Rotation不为0时，变换关系会更复杂，具体请参考12.4节。

实例演示　下面通过实例演示方法InverseTransformDirection的使用。

```
using UnityEngine;
using System.Collections;

public class InverseTransformDirection_ts : MonoBehaviour
{
    void Start()
    {
        //转换前向量
```

12

```
        Vector3 world_v = new Vector3(10.0f, 20.0f, 30.0f);
        //transform绕y轴旋转90度，使transform坐标系与世界坐标系方向不一致
        transform.eulerAngles = new Vector3(0.0f, 90.0f, 0.0f);
        Vector3 local_v = transform.InverseTransformDirection(world_v);
        //打印transform的position，方法InverseTransformDirection与transform的position无关
        Debug.Log("transform position:" + transform.position);
        //打印transform的lossyScale，方法InverseTransformDirection与transform的lossyScale无
            关
        Debug.Log("transform lossyScale:" + transform.lossyScale);
        Debug.Log("world_v:" + world_v);
        Debug.Log("local_v:" + local_v);
    }
}
```

在这段代码的Start方法中，首先声明了一个Vector3变量world_v，然后修改transform的欧拉角，使得transform自身坐标系与世界坐标系的x轴及z轴的方向不一致，接着调用方法InverseTransformDirection，将向量world_v从世界坐标系变换到transform的局部坐标系中，最后打印出相关信息，结果如图12-12所示，对于结果的解释如代码中注释所述。

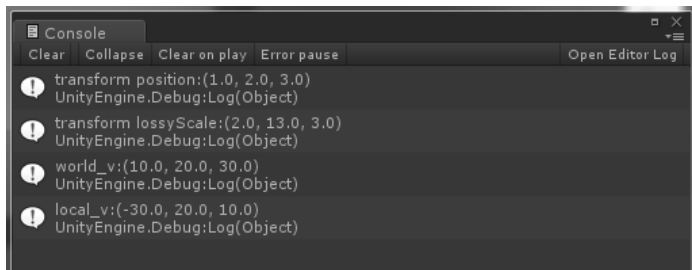


图12-12 InverseTransformDirection实例演示的运行结果

12.2.4 InverseTransformPoint方法：点的相对坐标向量

基本语法 (1) public Vector3 InverseTransformPoint(Vector3 position);

(2) public Vector3 InverseTransformPoint(float x, float y, float z);

功能说明 此方法用于返回参数position向量相对于GameObject对象局部坐标系的差向量，即返回向量position和向量transform.position的差值。例如，设A为Vector3实例，且transform的x、y、z轴方向和世界坐标系方向一致，有以下代码：

```
Vector3 B = transform.InverseTransformPoint(A);
```

则分量值B.x= (A.x−transform.position.x)/transform.lossyScale.x，B.y、B.z的值与此类似。更通俗的表达就是B= (A−transform.position)/transform.lossyScale。当transform的x、y、z轴方向和世界坐标系方向不一致，即当transform的Rotation不为0时，变换

关系会更复杂，具体请参考12.4节。

实例演示　下面通过实例演示方法InverseTransformPoint的使用。

```
using UnityEngine;
using System.Collections;
public class InverseTransformPoint_ts : MonoBehaviour {
    Vector3 A = new Vector3(50.0f, 30.0f, 10.0f);
    Vector3 B = Vector3.zero;
    void Start()
    {
        B = transform.InverseTransformPoint(A);
        Debug.Log("transform.position:" + transform.position);
        Debug.Log("transform.lossyScale:" + transform.lossyScale);
        Debug.Log("A:" + A);
        Debug.Log("B:" + B);
    }
}
```

在这段代码中，首先初始化了两个Vector3变量A和B，然后在Start方法中调用方法InverseTransformPoint求向量A相对于transform自身坐标系的差向量，并将返回值赋给B，最后打印出相关信息，如图12-13所示。由输出结果可知有以下运算关系：

B=(A-transform.position)/transform.lossyScale,

例如B.z=(A.z−transform.position.z)/transform.lossyScale.z，即3.3=(10.0−0.0)/3.0，具体请参考功能说明部分。

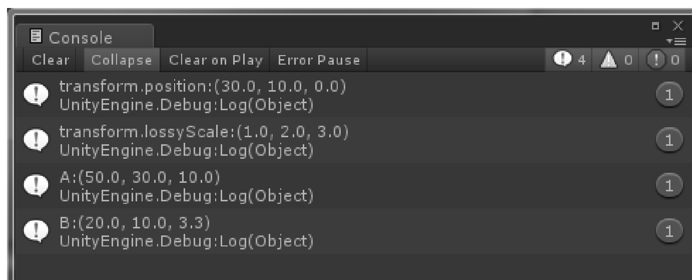


图12-13　InverseTransformPoint实例演示的运行结果

12.2.5　IsChildOf方法：是否为子物体

基本语法　`public bool IsChildOf(Transform parent);`

其中参数parent为父物体的Transform实例。

功能说明　此方法用于判断transform对应的GameObject对象是否为参数parent的子物体。例如，设A、B均是Transform实例，则A.isChildOf(B)在以下3种情况下都返回true值。

- A和B指向同一对象；

- ❑ A是B的一级子物体；
- ❑ A是B的多级子物体。

实例演示　下面通过实例演示方法IsChildOf的使用。在本实例演示中，首先需要在场景中创建3个Cube对象，它们的层级关系如图12-14所示，然后将如下脚本附加到Cube1物体上。

```
using UnityEngine;
using System.Collections;
public class IsChildOf_ts : MonoBehaviour
{
    public Transform t1, t2, t3;
    void Start()
    {
        Debug.Log("t1是否为其自身的子类：" + t1.IsChildOf(t1));
        Debug.Log("t2是否为t1的子类：" + t2.IsChildOf(t1));
        Debug.Log("t3是否为t1的子类：" + t3.IsChildOf(t1));
    }
}
```

在这段代码中，首先声明了3个Transform变量t1、t2和t3，分别指向场景中的物体Cube1、Cube2和Cube3,然后在Start方法中调用方法IsChildOf，分别检验t1所指向的物体是否为变量t1、t2和t3所指向物体的子类，输出结果如果12-15所示。

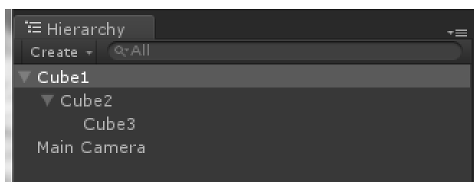


图12-14　场景中物体间层级关系

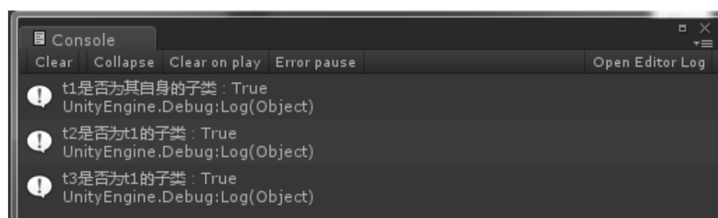


图12-15　IsChildOf实例演示运行结果

12.2.6　LookAt方法：物体朝向

基本语法　(1) public void LookAt(Transform target);

(2) public void LookAt(Vector3 worldPosition);

(3) public void LookAt(Transform target, Vector3 worldUp);

(4) public void LookAt(Vector3 worldPosition, Vector3 worldUp);

其中参数target为transform自身坐标系中z轴指向的目标，参数worldUp为transform自身坐标系中y轴最大限度指向的方向。

功能说明 此方法的功能是使得GameObject对象自身坐标系中的z轴指向target，y轴方向最大限度地指向worldUp方向。worldUp指的是世界坐标系中的方向。此方法通常被用在Camera上，使得Camera的forward看向目标target。worldUp默认情况下等于Vector3.up，即类似于一个人笔直地站在地面看向前方。若自定义worldUp的方向，则GameObject对象的forward方向一直指向target，然后transform绕着自身坐标系的z轴即forward方向旋转到一个使得自身的y轴方向最接近worldUp的地方。

实例演示 下面通过实例演示方法LookAt的使用。在本实例中，使一个SpotLight始终LookAt一个物体，在物体发生移动时SpotLight的方向会跟着调整，下述脚本附加在SpotLight上。

```
using UnityEngine;
using System.Collections;

public class LookAt_ts : MonoBehaviour
{
    public Transform tr_look;
    void Update()
    {
        transform.LookAt(tr_look);
        tr_look.RotateAround(Vector3.zero, Vector3.up, 60.0f * Time.deltaTime);
    }
}
```

在这段代码中，首先声明一个Transform类型变量tr_look，用于指向场景中的物体，然后在Update方法中，让SpotLight一直LookAt物体tr_look，并让物体tr_look绕原点旋转。请自行运行程序查看。

12.2.7 Rotate方法：绕坐标轴旋转

基本语法 (1) public void Rotate(Vector3 eulerAngles);

(2) public void Rotate(Vector3 eulerAngles, Space relativeTo);

(3) public void Rotate(float xAngle, float yAngle, float zAngle);

(4) public void Rotate(float xAngle, float yAngle, float zAngle, Space relativeTo);

其中参数eulerAngles为transform要旋转的欧拉角，参数relativeTo为transform旋转时参考的坐标系，默认为Space.Self。

功能说明 此方法的功能是使得transform实例在相对参数relativeTo的坐标系中旋转欧拉角eulerAngles。

实例演示　下面通过实例演示方法Rotate的使用。

```
using UnityEngine;
using System.Collections;

public class Rotate1_ts : MonoBehaviour {
    void Start () {
        //设置transform的欧拉角，使得transform自身坐标系和世界坐标系y轴方向不一致
        transform.eulerAngles = new Vector3(30.0f, 0.0f, 0.0f);
        transform.Rotate(new Vector3(0.0f,45.0f,0.0f),Space.Self);
        Debug.Log("Space.Self:" + transform.eulerAngles);

        //设置transform的欧拉角，使得transform自身坐标系和世界坐标系y轴方向不一致
        transform.eulerAngles = new Vector3(30.0f, 0.0f, 0.0f);
        transform.Rotate(new Vector3(0.0f, 45.0f, 0.0f), Space.World);
        Debug.Log("Space.World:" + transform.eulerAngles);

        //设置transform的欧拉角，使得transform自身坐标系和世界坐标系y轴方向不一致
        transform.eulerAngles = new Vector3(30.0f, 0.0f, 0.0f);
        transform.Rotate(new Vector3(0.0f, 45.0f, 0.0f));
        Debug.Log("默认坐标系："+transform.eulerAngles);
  }
}
```

在这段代码的Start方法中，首先使transform沿x轴旋转30度，使得transform自身坐标系和世界坐标系y轴方向不一致，然后调用Rotate方法使transform分别沿着自身坐标系、世界坐标系和默认坐标系的y轴旋转角45度，并打印出旋转后的transform欧拉角，结果如图12-16所示。

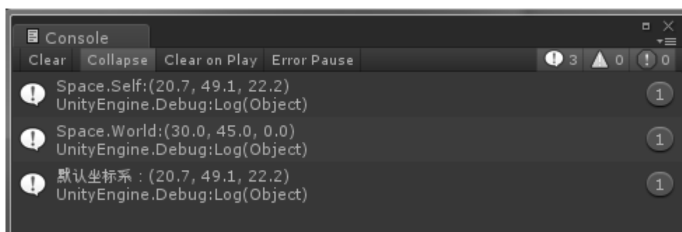


图12-16　Rotate实例演示的运行结果

12.2.8　Rotate方法：绕某个向量旋转

基本语法　(1) public void Rotate(Vector3 axis, float angle);

(2) public void Rotate(Vector3 axis, float angle, Space relativeTo);

其中参数axis为旋转轴方向，参数angle为旋转角度，参数relativeTo为参考坐标系，默认为Space.self。

功能说明　此方法的功能是使得GameObject对象在relativeTo坐标系中绕轴向量axis旋转angle

度。若想使GameObject对象实例绕着某个物体旋转，请用Transform.RotateAround(point : Vector3, axis : Vector3, angle : float)方法。

实例演示 下面通过实例演示方法Rotate的使用。

```
using UnityEngine;
using System.Collections;

public class Rotate2_ts : MonoBehaviour {
    void Start()
    {
        //设置transform的欧拉角，使得transform自身坐标系和世界坐标系y轴方向不一致
        transform.eulerAngles = new Vector3(30.0f, 0.0f, 0.0f);
        transform.Rotate(Vector3.up,45.0f, Space.Self);
        Debug.Log("绕自身坐标系的y轴方向旋转45度:" + transform.eulerAngles);

        //设置transform的欧拉角，使得transform自身坐标系和世界坐标系y轴方向不一致
        transform.eulerAngles = new Vector3(30.0f, 0.0f, 0.0f);
        transform.Rotate(Vector3.up, 45.0f, Space.World);
        Debug.Log("绕世界坐标系的y轴方向旋转45度:" + transform.eulerAngles);

        //设置transform的欧拉角，使得transform自身坐标系和世界坐标系y轴方向不一致
        transform.eulerAngles = new Vector3(30.0f, 0.0f, 0.0f);
        transform.Rotate(Vector3.up,45.0f);
        Debug.Log("绕默认坐标系的y轴方向旋转45度：" + transform.eulerAngles);
    }
}
```

在这段代码的Start方法中，首先使transform沿x轴旋转30度，使得transform自身坐标系和世界坐标系y轴方向不一致，然后调用Rotate方法使transform分别沿着自身坐标系、世界坐标系和默认坐标系的y轴旋转角45度，并打印出旋转后的transform欧拉角，结果如图12-17所示。

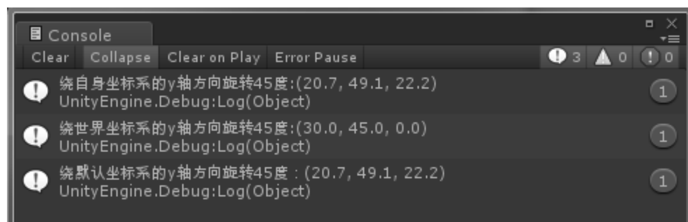


图12-17 Rotate实例演示的运行结果

12.2.9 RotateAround方法：绕轴点旋转

基本语法 (1) public void RotateAround(Vector3 axis, float angle);

(2) public void RotateAround(Vector3 point, Vector3 axis, float angle);

其中参数point为参考点坐标，参数axis为旋转轴方向，参数angle为旋转角度。

功能说明　此方法的功能是使得GameObject对象绕着point点的axis方向旋转angle度。

实例演示　下面通过实例演示方法RotateAround的使用。

```
using UnityEngine;
using System.Collections;

public class RotateAround_ts : MonoBehaviour {
    public Transform point_r;
        void Update () {
        transform.RotateAround(point_r.position ,Vector3.up,30.0f*Time.deltaTime);
  }
}
```

在这段代码中，首先声明了一个Transform类型变量point_r，然后在Update方法中调用transform的RotateAround方法，使得transform对应GameObject对象绕着point_r所在位置的Vector3.up方向以每帧30.0f*Time.deltaTime的角度做圆周运动，请自行运行程序查看。

12.2.10　TransformDirection方法：坐标系转换

基本语法　(1) public Vector3 TransformDirection(Vector3 direction);

其中参数direction为待转换的Vector3实例向量。

(2) public Vector3 TransformDirection(float x, float y, float z);

功能说明　此方法用于将向量direction从transform局部坐标系转换到世界坐标系。例如，设local_v为Vector3实例，且transform的x、y、z轴方向和世界坐标系的x、y、z轴方向保持一致，有以下代码：

```
Vector3 world_v = transform.TransformDirection(local_v);
```

则world_v.x=local_v.x，world_v.y、world_v.z的值以此类推，即转换后的向量world_v只与transform的Rotation和向量local_v有关，而与transform的position和scale值无关，此为与方法TransformPoint (position : Vector3) 功能不同的地方。当transform的x、y、z轴方向和世界坐标系方向不一致，即当transform的Rotation不为0时，变换关系会更复杂，具体请参考12.4节。

实例演示　下面通过实例演示方法TransformDirection的使用。

```
using UnityEngine;
using System.Collections;

public class TransformDirection_ts : MonoBehaviour
{
    void Start()
    {
         Vector3 local_v = new Vector3(1.0f, 2.0f, 3.0f);
```

```
            transform.eulerAngles = new Vector3(0.0f, 90.0f, 0.0f);
            //将local_v从局部坐标系转到世界坐标系中
            Vector3 world_v = transform.TransformDirection(local_v);
            //transform的position值不影响变换结果
            Debug.Log("transform position:" + transform.position);
            //transform的lossyScale值不影响变换结果
            Debug.Log("transform lossyScale:" + transform.lossyScale);
            Debug.Log("local_v:" + local_v);
            Debug.Log("world_v:" + world_v);
        }
    }
```

在这段代码的Start方法中，首先声明了一个Vector3变量local_v，然后修改transform的欧拉角，使得transform自身坐标系与世界坐标系的x轴及z轴方向不一致，接着调用方法TransformDirection将local_v从transform局部坐标系变换到世界坐标系，最后打印出相关信息，结果如图12-18所示，对于结果的解释如代码中注释所述。

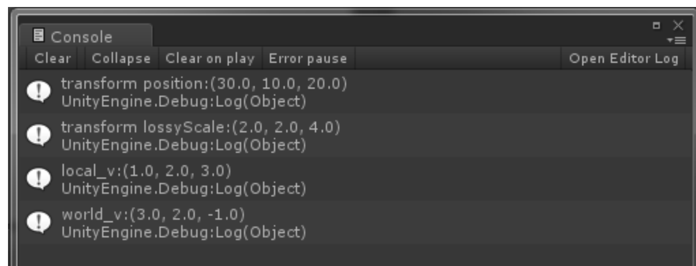


图12-18 TransformDirection实例演示的运行结果

12.2.11 TransformPoint方法：点的世界坐标位置

基本语法 (1) public Vector3 TransformPoint(Vector3 position);

其中参数position为transform局部坐标系的向量。

(2) public Vector3 TransformPoint(float x, float y, float z);

功能说明 此方法用于返回GameObject对象局部坐标系中向量position在世界坐标系中的位置。例如，设A为Vector3，且transform的x、y、z轴方向和世界坐标系方向一致，有以下代码：

```
Vector3 B = transform.TransformPoint(A);
```

则分量值B.x=transform.position.x+A.x* transform.lossyScale.x，B.y、B.z的值与此类似。更通俗的表达就是：B=transform.position + A* transform.lossyScale。当transform的x、y、z轴方向和世界坐标系方向不一致，即当transform的Rotation不为0时，变换关系会更复杂，具体请参考12.4节。

实例演示 下面通过实例演示方法TransformPoint的使用。

```
using UnityEngine;
using System.Collections;
public class TransformPoint_ts : MonoBehaviour {
    Vector3 A=new Vector3(1.0f,2.0f,3.0f);
    Vector3 B=Vector3.zero;
    void Start () {
        B = transform.TransformPoint(A);
        Debug.Log("transform.position:" + transform.position);
        Debug.Log("transform.lossyScale:" + transform.lossyScale);
        Debug.Log("A:"+A);
        Debug.Log("B:"+B);
    }
}
```

在这段代码中，首先初始化了两个Vector3变量A和B，然后在Start方法中调用方法TransformPoint将向量A转换到transform的自身坐标系中，并将返回值赋给B，最后打印出相关信息，如图12-19所示。由输出结果可知有如下关系：

B=transform.position+transform.lossyScale*A,

例如B.x= transform.position.x+ transform.lossyScale.x*A.x，即32.0=30.0+2.0*1.0，具体请参考功能说明部分。

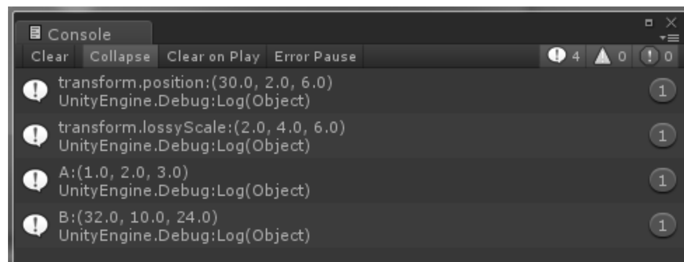


图12-19 TransformPoint实例演示的运行结果

12.2.12 Translate 方法：相对坐标系移动

基本语法 (1) `public void Translate(Vector3 translation);`

(2) `public void Translate(Vector3 translation, Space relativeTo);`

(3) `public void Translate(float x, float y, float z);`

(4) `public void Translate(float x, float y, float z, Space relativeTo);`

其中参数translation为移动向量，包括方向和大小，参数relativeTo为参考坐标系空间，默认为Space.Self。

功能说明 此方法的功能是使得GameObject对象在参数relativeTo的坐标系空间中移动参数translation指定的向量。

实例演示 下面通过实例演示方法Translate的使用。

```
using UnityEngine;
using System.Collections;

public class Translate1_ts : MonoBehaviour {
    void Start () {
        //沿y轴旋转30度，使得transform自身坐标系和世界坐标系x轴方向不一致
        transform.eulerAngles = new Vector3(0.0f, 30.0f, 0.0f);
        //坐标位置归零
        transform.position = Vector3.zero;
        transform.Translate(new Vector3(10.0f, 0.0f, 0.0f), Space.Self);
        Debug.Log("沿自身坐标系x轴移动10个距离：" + transform.position);
        //坐标位置归零
        transform.position = Vector3.zero;
        transform.Translate(new Vector3(10.0f, 0.0f, 0.0f), Space.World);
        Debug.Log("沿世界坐标系x轴移动10个距离：" + transform.position);
        //坐标位置归零
        transform.position = Vector3.zero;
        transform.Translate(new Vector3(10.0f, 0.0f, 0.0f));
        Debug.Log("沿默认坐标系x轴移动10个距离：" + transform.position);
    }
}
```

在这段代码的Start方法中，首先使transform沿y轴旋转30度，使得transform自身坐标系和世界坐标系x轴方向不一致，然后调用Translate方法，使transform分别沿着自身坐标系、世界坐标系和默认坐标系移动向量Vector(10.0f, 0.0f, 0.0f)，并打印出移动后的transform位置，结果如图12-20所示。

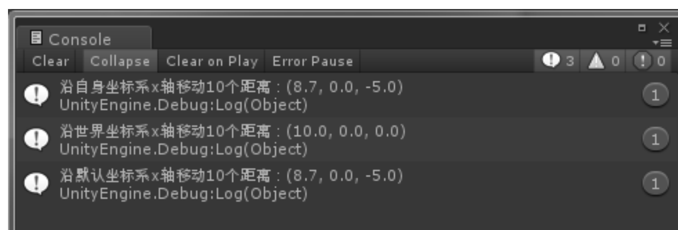


图12-20 Translate实例演示的运行结果

12.2.13 Translate方法：相对其他物体移动

基本语法 (1) `public void Translate(Vector3 translation, Transform relativeTo);`

(2) `public void Translate(float x, float y, float z, Transform relativeTo);`

其中参数translation为移动向量，包括方向和大小，参数relativeTo为移动参考物体，默认为Space.World。

功能说明 此方法的功能是使得GameObject对象在相对relativeTo的坐标系中移动向量translation。

实例演示　下面通过实例演示方法Translate的使用。

```
using UnityEngine;
using System.Collections;

public class Translate2_ts : MonoBehaviour {
    public Transform relativeTo;
    void Start () {
        //将relativeTo沿y轴旋转30度，使得relativeTo自身坐标系和世界坐标系x轴方向不一致
        relativeTo.eulerAngles = new Vector3(0.0f, 30.0f, 0.0f);
        //transform坐标位置归零
        transform.position = Vector3.zero;
        transform.Translate(new Vector3(10.0f, 0.0f, 0.0f), relativeTo);
        Debug.Log("沿relativeTo坐标系x轴移动10个距离：" + transform.position);

        //坐标位置归零
        transform.position = Vector3.zero;
        transform.Translate(new Vector3(10.0f, 0.0f, 0.0f));
        Debug.Log("沿默认坐标系x轴移动10个距离：" + transform.position);
    }
}
```

在这段代码中，首先声明了一个Transform类型变量relativeTo，然后在Start方法中修改relativeTo的欧拉角，使其与世界坐标系方向不一致，最后调用transform的Translate方法，分别在相对relativeTo坐标系和默认坐标系中移动向量Vector3(10.0f, 0.0f, 0.0f)，并打印出结果，如图12-21所示。

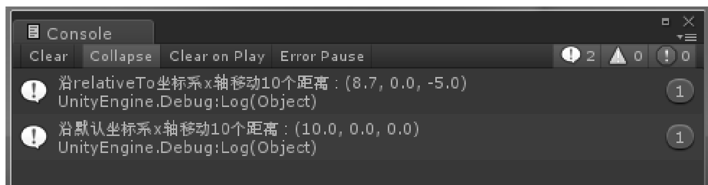


图12-21　Translate实例演示的运行结果

12.3　关于 localScale 和 lossyScale 的功能注解

对属性localScale和lossyScale的不当使用往往会使得GameObject对象产生错误的变形，对它们的变换及使用说明如下。

- 当GameObject对象A为GameObject对象B的父物体时，父物体A的各个分量放缩值x、y、z的大小应该保持1：1：1的比例，否则当子物体B的Rotation值比例不为1：1：1时，B物体将会发生变形。图12-22为将父物体A的localScale值设为Vector3(4.0f,1.0f,1.0f)，子物体B的欧拉角为Vector（0.0f,0.0f,0.0f）时的状态。图12-23为将B的欧拉角变为Vector3（0.0f,40.0f,0.0f）后的状态，此时B的形状已经发生了严重的变形。

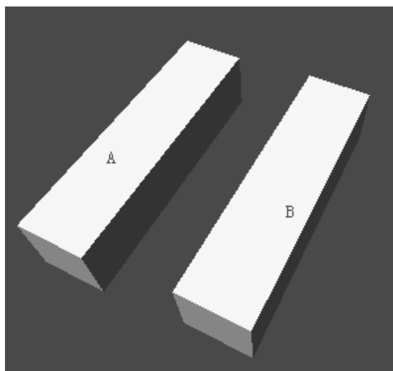


图12-22　物体B未发生旋转时

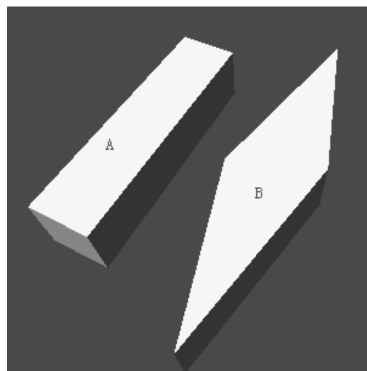


图12-23　物体B发生旋转后

- 设GameObject对象A为B的父物体，当A物体各个分量的放缩值保持1∶1∶1的比例时，子物体B的lossyScale返回值即为B物体相对世界坐标系的放缩值，关系为

B.localScale= B.lossyScale/A.localScale

实例演示　下面通过实例演示属性localScale和lossyScale的关系。

```
using UnityEngine;
using System.Collections;
public class LocalScaleOrLossyScale_ts : MonoBehaviour
{
    void Start()
    {
        Debug.Log("父物体A放缩值：" + transform.parent.localScale);
        Debug.Log("子物体B的全局有损放缩值：" + transform.lossyScale);
        Debug.Log("子物体B的局部放缩值：" + transform.localScale);
        //检验本注解部分的算法描述是否正确
        if (transform.localScale.x == (transform.lossyScale.x / transform.parent.localScale.x)
            &&
            transform.localScale.y == (transform.lossyScale.y / transform.parent.localScale.y)
            &&
            transform.localScale.z == (transform.lossyScale.z / transform.parent.localScale.z))
        {
            Debug.Log("True");
        }
        else
        {
            Debug.Log("False");
        }
    }
}
```

在这段代码的Start方法中，首先分别打印出父物体A的放缩值、物体B的全局有损放缩值和物体B的局部放缩值，然后使用transform的各个分量值检验注解部分的算法描述是否正确，程序输出结果如图12-24所示。

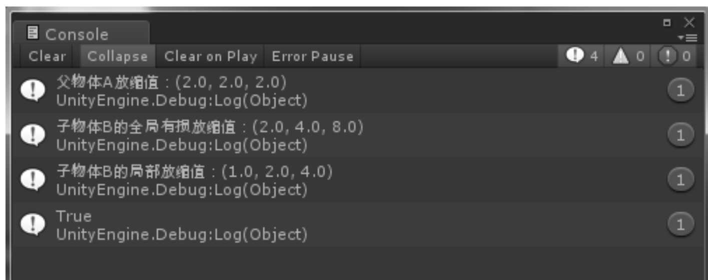


图12-24　实例演示运行结果

12.4　关于 Transform 类中涉及空间变换的几个属性和方法的功能注解

在Transform类中，涉及空间变换的属性和方法主要有worldToLocalMatrix、localToWorldMatrix、TransformPoint、InverseTransformPoint、TransformDirection和InverseTransformDirection，下面对它们之间的联系及各自的不同之处进行说明。

本注解约定　由于这几个属性和方法都涉及世界坐标系和局部坐标系的变换，在绘制坐标系时约定世界坐标系的x、y、z轴分别标注为wx、wy、wz，局部坐标系的x、y、z轴分别标注为lx、ly、lz，当世界坐标系和局部坐标系的某个轴向相同时以括号标注，比如它们的y轴方向同向时便标注为wy(ly)。

功能注解

- worldToLocalMatrix和localToWorldMatrix属于Transform类的只读属性。
- 利用worldToLocalMatrix和localToWorldMatrix的返回值可以实现与方法TransformPoint和InverseTransformPoint类似的功能。worldToLocalMatrix、localToWorldMatrix、TransformPoint和InverseTransformPoint在空间转换方式上相似，都会受到局部坐标系即transform自身坐标系的位置（Position）和放缩值（Scale）的影响。而方法TransformDirection和InverseTransformDirection在空间转换方式上相似，它们不会受到局部坐标系即transform自身坐标系的位置和放缩值的影响。
- 这几个属性或方法在空间转换时都会受到局部坐标系即transform自身坐标系中Rotation的影响，但它们各自受影响的方式不同。下面以GameObject对象仅绕Y轴旋转为例解释。

TransformDirection的转换关系如图12-25和图12-26所示，执行代码如下。

```
W=transform. TransformDirection(L);
```

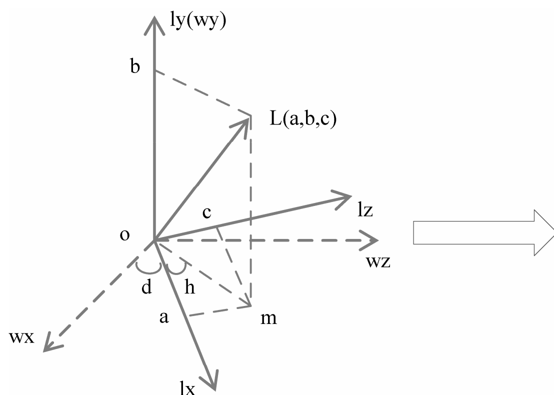


图12-25　TransformDirection转换前

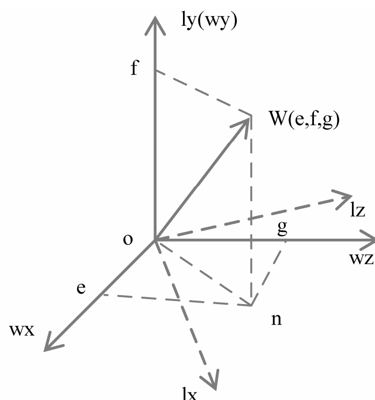


图12-26　TransformDirection转换后

图12-25中L(a,b,c)是GameObject对象局部坐标系中的坐标，d是GameObject对象绕y轴旋转的角度，h是向量oL在xoz平面的投影向量om和lx轴的夹角。图12-26中W(e,f,g)是L点执行转换后在世界坐标系中的位置，则有如下关系：

模长|oL|=|oW|;

分量值e=a*cos(d+h)/cos(h);

分量值f=b;

分量值g=c*sin(d+h)/sin(h);

InverseTransformDirection的转换关系如图12-27和图12-28所示，执行代码如下：

```
L=transform. InverseTransformDirection(W);
```

图12-27中W(a,b,c)是在世界坐标系中的某个坐标，d是GameObject对象绕Y轴旋转的角度，h是向量oW在xoz平面的投影om和lx轴的夹角。图12-28中L(e,f,g)是W点执行转换后在GameObject对象局部坐标系中的位置，则有如下关系：

模长|oW|=|oL|;

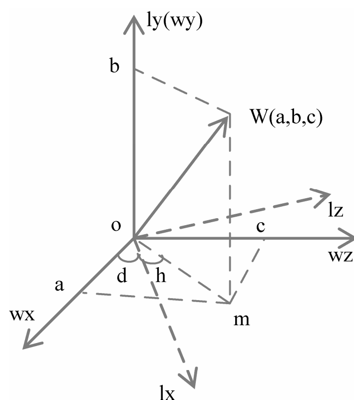
分量值e=a*cos(h)/cos(d+h);

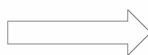
分量值f=b;

分量值g=c*sin(h)/sin(d+h);

TransformPoint的转换关系如图12-29和图12-30所示，执行如下代码：

```
W=transform. TransformPoint(L);
```





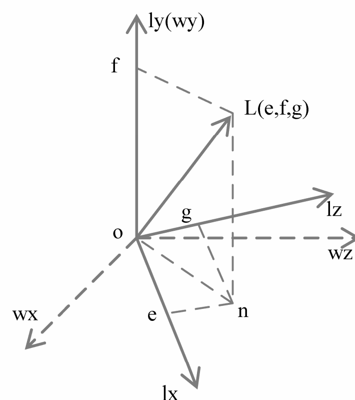


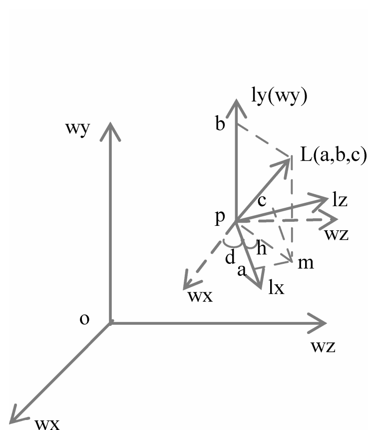
图12-27　InverseTransformDirectio转换前　　　图12-28　InverseTransformDirection转换后

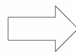
图12-29中L(a,b,c)是GameObject对象局部坐标系中的坐标，p是GameObject对象的坐标位置Position，d是GameObject对象绕Y轴旋转的角度，h是向量oL在xoz平面的投影向量om和lx轴的夹角。图12-30中W(e,f,g)是L点执行转换后在世界坐标系中的位置，则有如下关系：

分量值e=p.x+a*transform.lossyscale*cos(d);

分量值f=p.y+b*transform.lossyscaley;

分量值g=p.z+c*transform.lossyscale*cos(d);





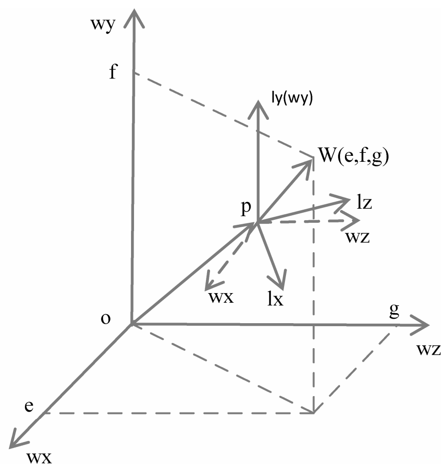
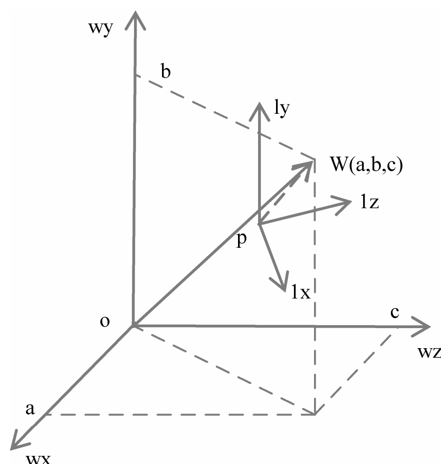
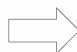


图12-29　TransformPoint转换前　　　图12-30　TransformPoint转换后

InverseTransformPoint的转换关系如图12-31和图12-32所示，执行如下代码：

```
L=transform. InverseTransformPoint(W);
```





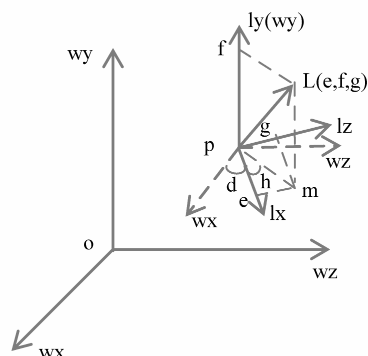


图12-31 InverseTransformPoint转换前　　图12-32 InverseTransformPoint转换后

图12-31中W(a,b,c)是世界坐标系中的坐标。图12-32中L(e,f,g)是W点执行转换后在GameObject对象局部坐标系中的位置，p是GameObject对象的世界坐标位置Position，d是GameObject对象绕Y轴旋转的角度，h是向量oL在xoz平面的投影向量om和lx轴的夹角。则有如下关系：

e=((a–p.x)/transform.lossyScale.x)*cos(d);

f=(b–p.y)/transform.lossyScale.y;

g=((c–p.z)/transform.lossyScale.z)*cos(d);

由于worldToLocalMatrix和InverseTransformPoint 转换方式类似，localToWorldMatrix和TransformPoint转换方式类似，在此对localToWorldMatrix和worldToLocalMatrix的转换方式就不再赘述。

12

第 13 章 Vector2类

Vector2类是Unity中用来存储二维向量或二维点坐标的结构体类型。本章主要介绍了Vector2类的一些实例方法、静态方法和运算符。

13.1　Vector2 类实例方法

在Vector2类中，涉及的实例方法只有Normalize方法。由于Vector2的实例属性normalized与此方法功能相近，于是将它们放到一起介绍。

Normalize方法：单位化Vector2 实例

基本语法　public void Normalize();

功能说明　此方法用来单位化向量，即将Vector2实例进行单位化处理。此方法改变了原始向量，无返回值。实例属性normalized与此方法功能相同，但使用属性normalized来单位化向量时，不改变原始向量值，且有返回值，请参考实例演示。

实例演示　下面通过实例演示方法Normalize的使用。

```
using UnityEngine;
using System.Collections;

public class Normalize_ts : MonoBehaviour {
    void Start()
    {
        Vector2 v1 = new Vector2(1.0f, 2.0f);
        //使用属性来单位化，不改变原始向量值，有返回值
        Vector2 v2 = v1.normalized;
        Debug.Log("使用属性v2.normalized后v1的值：" + v1);
        Debug.Log("使用属性v2.normalized后的返回值v2：" + v2);
        //重置向量v1
        v1.Set(1.0f, 2.0f);
        //使用实例方法改变原始值，无返回值
        v1.Normalize();
        Debug.Log("使用实例方法v1.Normalize后v1的值：" + v1);
    }
}
```

在这段代码的Start方法中，首先初始化了一个Vector2向量v1，然后分别使用实例属性normalized和实例方法Normalize对v1进行单位化，并将单位化后的结果打印出来，如图13-1所示，对结果的解释请参考代码注释。

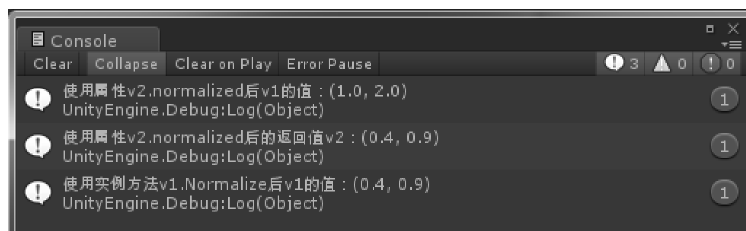


图13-1　Normalize实例演示的运行结果

13.2　Vector2 类静态方法

在Vector2类中，涉及的静态方法有Angle方法、ClampMagnitude方法、Lerp方法、MoveTowards 方法和Scale方法，由于静态方法Scale和Vector2实例方法中的Scale方法功能相近，于是将它们放到一起介绍。下面将详细介绍这些方法。

13.2.1　Angle方法：两个向量夹角

基本语法　public static float Angle(Vector2 from, Vector2 to);

其中参数from为起始向量，参数to为结束向量。

功能说明　此方法用于返回两个Vector2实例的夹角，单位为角度，返回值的取值范围为[0,180]，并且当from和to中至少有一个向量为Vector2.zero的时候返回值为90。

实例演示　下面通过实例演示Angle方法的使用。

```
using UnityEngine;
using System.Collections;

public class Angle_ts : MonoBehaviour
{
    void Start()
    {
        Debug.Log("1:" + Vector2.Angle(new Vector2(1.0f, 0.0f), new Vector2(-1.0f, 0.0f)));
        Debug.Log("2:" + Vector2.Angle(new Vector2(1.0f, 0.0f), new Vector2(-1.0f, -1.0f)));
        Debug.Log("3:" + Vector2.Angle(new Vector2(0.0f, 0.0f), new Vector2(-1.0f, 0.0f)));
    }
}
```

在这段代码的Start方法中，依次打印出了3种不同情况下Vector静态方法Angle的返回值，输出结果如图13-2所示。

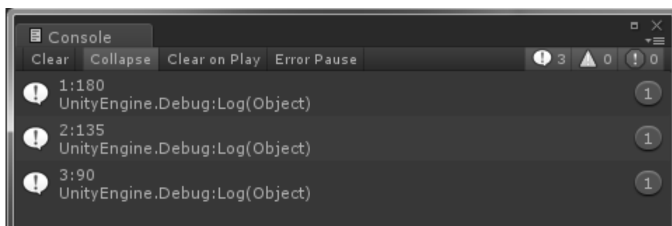


图13-2　方法Angle实例演示的运行结果

13.2.2　ClampMagnitude方法：向量长度

基本语法　public static Vector2 ClampMagnitude(Vector2 vector, float maxLength);

功能说明　此方法用于返回向量的长度，且最大不超过maxLength，对其使用说明如下。

- 设有代码：

```
Vector2 A = new Vector2(ax, ay);
float b=maxL;
Vector2 C = Vector2.ClampMagnitude(A, b);
```

　其中ax、ay和maxL为已知数值，则：

$$C.x=ax*K;$$

$$C.y=ay*K;$$

　其中 $K = \left|\frac{b}{\sqrt{ax^2+ay^2}}\right| > 1?1:\frac{b}{\sqrt{ax^2+ay^2}}$ 。

　当A为Vector2.zero时，返回值为二维零向量。

- 此方法功能类似一个钟摆的摆动，|ax|代表沿着x轴的最大摆幅，|ay|代表沿着y轴的最大摆幅，而b值则代表着摆动的幅度。

实例演示　下面通过实例演示方法ClampMagnitude的使用。

```
using UnityEngine;
using System.Collections;

public class ClampMagnitude_ts : MonoBehaviour
{
    void Start()
    {
        Vector2 A = new Vector2(10.0f, 20.0f);
        //K值大于1时返回A的初始值
        float b = A.magnitude * 3.0f;
        Vector2 C = Vector2.ClampMagnitude(A, b);
        Debug.Log("K值大于1时返回A的初始值:" + C.ToString());
```

```
        //K值小于-1时返回A的初始值
        b = -A.magnitude * 3.0f;
        C = Vector2.ClampMagnitude(A, b);
        Debug.Log("K值小于-1时返回A的初始值:" + C.ToString());
        //K值介于-1和1之间时按算法求值
        b = A.magnitude * 0.7f;
        C = Vector2.ClampMagnitude(A, b);
        Debug.Log("K值介于-1和1之间时时按算法求值:" + C.ToString());
        //当A为零向量时返回值永远为零向量
        A.Set(0.0f,0.0f);
        b = A.magnitude * 0.7f;
        C = Vector2.ClampMagnitude(A, b);
        Debug.Log("当A为零向量时返回值永远为零向量:" + C.ToString());
    }
}
```

在这段代码中，首先初始化了一个Vector2变量A，然后分别演示了当K取不同范围值时方法ClampMagnitude的返回值，K的计算方法请参考功能说明部分，程序运行结果如图13-3所示，对结果的解释请参考代码注释。

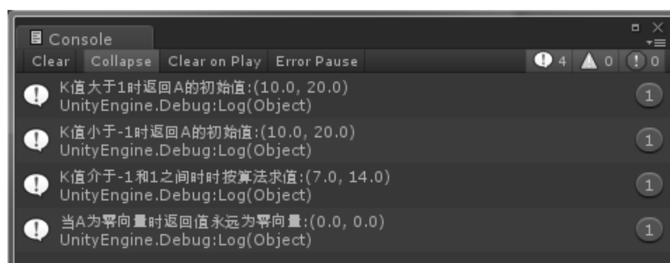


图13-3 ClampMagnitude实例演示的运行结果

13.2.3 Lerp方法：向量插值

基本语法 public static Vector2 Lerp(Vector2 from, Vector2 to, float t);

其中参数from为插值的起始向量，参数to为插值的结束向量，参数t为插值系数。

功能说明 此方法用于求从参数from到参数to的插值向量。例如，设有代码：

```
Vector2 v1=Vector2.Lerp(new Vector2(a,b),new Vector2(c,d),e);
```

则代码执行后v1的分量值分别为

$$v1.x=a*(1-e)+c*e;$$

$$v1.y=b*(1-e)+d*e;$$

其中e的有效范围为[0,1]，即

$$e= e<0\ ?\ 0 : e>1?\ 1 : e;$$

实例演示 下面通过实例演示方法Lerp的使用。

```
using UnityEngine;
using System.Collections;

public class Lerp_ts : MonoBehaviour
{
    void Start()
    {
        Vector2 v1 = new Vector2(4.0f, 12.0f);
        Vector2 v2 = new Vector2(9.0f, 20.0f);
        //e<0时返回值为v1的值
        Debug.Log("e<0时返回值为v1的值:" + Vector2.Lerp(v1, v2, -2.0f));
        //e>1时返回值为v2的值
        Debug.Log("e>1时返回值为v2的值:" + Vector2.Lerp(v1, v2, 2.0f));
        //e>0且e<1时按插值算法求返回值
        Debug.Log("e>0且e<1时按插值算法求返回值:" + Vector2.Lerp(v1, v2, 0.7f));
    }
}
```

在这段代码的Start方法中，首先声明了两个Vector2变量v1和v2，然后调用方法Lerp分别打印出当参数e<0、e>1和0<e<1时方法的返回值，输出结果如图13-4所示。

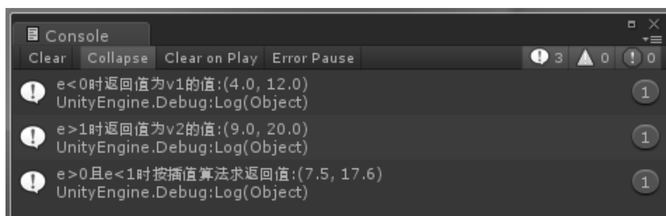


图13-4 Lerp实例演示的运行结果

13.2.4 MoveTowards方法：向量插值

基本语法 public static Vector2 MoveTowards(Vector2 current, Vector2 target, float maxDistanceDelta);

其中参数current为移动起始点坐标，参数target为移动目标点，参数maxDistanceDelta为移动的参考系数。

功能说明 此方法用于返回两个向量的插值，且最大插值不超过maxDistanceDelta，对其使用解释如下。

- 设有以下代码：

```
Vector2 A = new Vector2(ax, ay);
Vector2 B = new Vector2(bx, by);
float sp=maxDis;// maxDis为已知值
Vector2 C = Vector2.MoveTowards(A, B, sp);
```

其中ax、bx、ay、by和sp为已知值。则向量C的计算方法为

$$C=A+K*d=(ax+K*(bx-ax), ay+K*(by-ay));$$

其中，$d=(dx,dy)=B-A$，$K=\frac{sp}{\sqrt{dx^2+dy^2}}>1?1:\frac{sp}{\sqrt{dx^2+dy^2}}$。

- MoveTowards和Lerp方法的功能类似，但有一定的区别。同样是从一个向量A插值到向量B，MoveTowards要比Lerp的插值次数多，视觉效果上也更平滑，但是MoveTowards的计算更消耗时间。在我笔记本上的测试结果为：以万次计算MoveTowards的计算时间是Lerp的3倍多。

实例演示 下面通过实例演示方法MoveTowards的使用。

```
using UnityEngine;
using System.Collections;

public class MoveTowards_ts : MonoBehaviour
{
    void Start()
    {
        Vector2 A = new Vector2(10.0f, 20.0f);
        Vector2 B = new Vector2(30.0f, 40.0f);
        //当K=1时返回B的值
        float sp = (B - A).magnitude;
        Vector2 C = Vector2.MoveTowards(A, B, sp);
        Debug.Log("当K=1时返回B的值:" + C.ToString());
        //当K>1时返回B的值
        sp += 10.0f;
        C = Vector2.MoveTowards(A, B, sp);
        Debug.Log("当K>1时返回B的值:" + C.ToString());
        //当K=0时返回A的值
        sp = 0.0f;
        C = Vector2.MoveTowards(A, B, sp);
        Debug.Log("当K=0时返回A的值:" + C.ToString());
        //当K<0时按算法计算
        sp = -10.0f;
        C = Vector2.MoveTowards(A, B, sp);
        Debug.Log("当K<0时按算法计算:" + C.ToString());
        //当K>0且K<1时按算法计算
        sp = (B - A).magnitude * 0.7f;
        C = Vector2.MoveTowards(A, B, sp);
        Debug.Log("当K>0且K<1时按算法计算:" + C.ToString());
    }
}
```

在这段代码的Start方法中，首先初始化了两个Vector2变量A和B，然后分别演示了当K取不同范围值时方法MoveTowards的返回值，K的计算方法请参考功能说明部分，程序运行结果如图13-5所示，对结果的解释请参考代码注释。

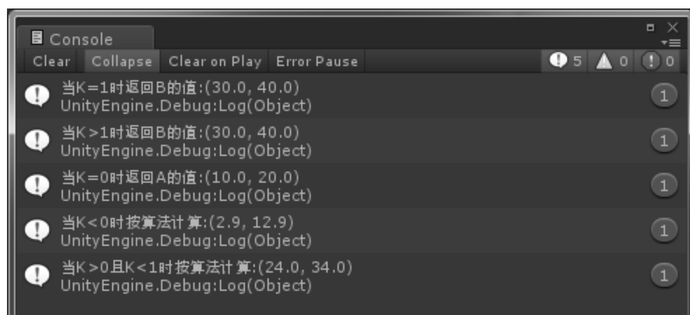


图13-5 MoveTowards实例演示的运行结果

13.2.5 Scale方法：向量放缩

基本语法 public static Vector2 Scale(Vector2 a, Vector2 b);

其中参数a、b为两个二维向量，不分前后次序。

功能说明 此方法用于返回向量a按向量b进行放缩后的值，例如，设Vector2 v2=Vector2.Scale(new Vector2(x1,y1),new Vector2(x2,y2))，则v2.x=x1*x2，v2.y=y1*y2，即求向量a和向量b的乘积。

提　　示 注意此方法和实例方法中Scale用法的区别，例如，设有代码：

```
Vector2 v2=new Vector2(x1,y1);
v2.scale(new Vector(x2,y2));
```

则v2现在的分量值分别为： v2.x==x1*x2; v2.y==y1*y2;

实例演示 下面通过实例演示方法Scale的使用。

```
using UnityEngine;
using System.Collections;

public class Scale_ts : MonoBehaviour
{
    void Start()
    {
        Vector2 v2 = new Vector2(1.0f, 2.0f);
        //实例方法会改变原始向量v2的值，无返回值
        v2.Scale(new Vector2(2.0f, 3.0f));
        Debug.Log("使用实例方法v2.Scale后v2值：" + v2);

        //使用Set方法重设v2
        v2.Set(1.0f, 2.0f);
        //静态方法不会改变原始向量的值，其返回值为放缩后的向量
        Vector3 v3 = Vector2.Scale(v2, new Vector2(4.0f, 6.0f));
        Debug.Log("使用静态方法Vector2.Scale后v2值：" + v2);
        Debug.Log("使用静态方法Vector2.Scale后v3值：" + v3);
```

```
    }
}
```

在这段代码中，首先初始化了一个Vector2变量v2，然后分别使用Vector2的实例方法Scale和静态方法Scale对v2进行放缩，并打印出相关信息，程序运行结果如图13-6所示，对运行结果的解释请参考代码注释。

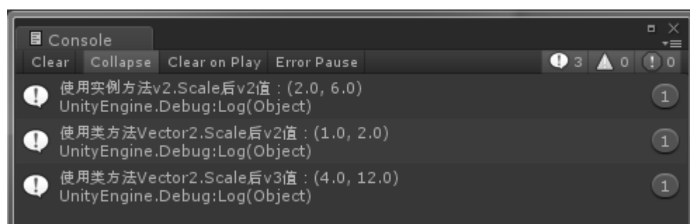


图13-6　Scale实例演示的运行结果

13.3　Vector2 类运算符

在Vector2类中，涉及的运算符有相等（“==”）运算符，下面简要介绍这个运算符。

operator == (lhs : Vector2, rhs : Vector2)

功能说明　此运算符用于判断两个向量lhs和rhs是否相等。不仅当lhs和rhs的分量值都相同时返回true，而且当lhs和rhs的各个分量值虽然不同但足够接近时也会返回true。经本机测试，当lhs和rhs的各个分量的小数点后六位相同或后五位相同第六位也足够接近的情况下返回true。例如：

当lhs=(1.123453,3.123453)，rhs=(1.123459,3.123459)时，lhs==rhs会返回true；

当lhs=(1.123451,3.123451)，rhs=(1.123459,3.123459)时，lhs==rhs会返回false。

实例演示　下面通过实例演示运算符“==”的使用。

```
using UnityEngine;
using System.Collections;

public class EqualOrNot_ts : MonoBehaviour
{
    void Start()
    {
        //A、B小数点后五位相等，第六位足够接近时返回true
        Vector2 A = new Vector2(1.123453f, 3.123453f);
        Vector2 B = new Vector2(1.123459f, 3.123459f);
        Debug.Log("First:" + (A == B));
        //A、B小数点后五位相等，但第六位不够接近时也可能返回false
        A = new Vector2(1.123451f, 3.123451f);
        Debug.Log("Second:" + (A == B));
        //A、B小数点后六位相等时一定返回true
        A = new Vector2(1.1234560f, 3.1234560f);
```

13

```
        B = new Vector2(1.1234569f, 3.1234569f);
        Debug.Log("Third:" + (A == B));
    }
}
```

在这段代码的Start方法中，分别演示了当向量A、B的各个分量值在不同接近程度的情况下，运算符“==”的返回值，程序运行输出结果如图13-7所示，对结果的解释请参考代码注释。

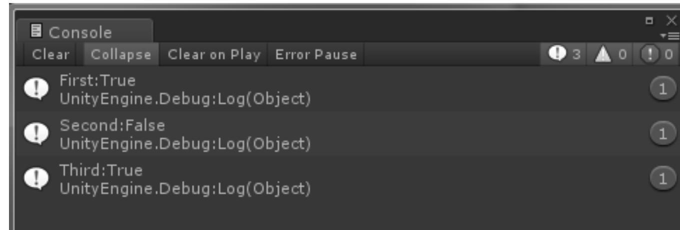


图13-7　实例演示运行结果

第 14 章

Vector3类

14

Vector3类属于结构体类型，用来表示Unity中的三维向量或三维坐标点。本章主要介绍了Vector3类的一些实例属性、实例方法、静态方法和运算符，并在本章最后对两组功能相近的API进行了注解。

14.1 Vector3 类实例属性

在Vector3类中，涉及的实例属性有normalized属性和sqrMagnitude属性。由于Vector3的实例方法Normalized()和实例属性normalized的功能相近，所以将它们放到一起介绍。下面详细介绍这些属性。

14.1.1 normalized属性：单位化向量

基本语法 public Vector3 normalized { get; }

功能说明 此属性用来获取Vector3实例的单位向量，即返回向量的方向与原向量方向相同，而模长变为1。注意此属性和实例方法Normalized()的区别。设A、C均为Vector3实例，则

- 执行代码C = A.normalized后只是将向量A的单位向量赋给向量C，而向量A自身未变。
- 执行代码A. Normalize()便会将向量A进行单位化处理，使得原向量A变成了单位向量。
- 执行代码C = Vector3.Normalize(A)的结果与执行代码C = A.normalized的相同，即只是将A的单位向量赋给了向量C，而向量A未被改变，因此编程中常用代码C = A.normalized代替。

实例演示 下面通过实例演示属性normalized和实例方法Normalized的使用。

```
using UnityEngine;
using System.Collections;

public class Normalized_ts : MonoBehaviour
{
```

```
    void Start()
    {
        Vector3 v1 = new Vector3(1.0f, 2.0f, 3.0f);
        Vector3 v2 = v1.normalized;
        //使用v2 = v1.normalized后v1的值不会改变，而v2的值为v1的单位向量
        Debug.Log("v2 = v1.normalized后v1的值：" + v1);
        Debug.Log("v2 = v1.normalized后v2的值：" + v2);
        //"v2 = Vector3.Normalize(v1)"与"v2 = v1.normalized"实现功能相同，但不常用
        v2 = Vector3.Normalize(v1);
        Debug.Log("v2 = Vector3.Normalize(v1)后v1的值：" + v1);
        Debug.Log("v2 = Vector3.Normalize(v1)后v2的值：" + v2);
        //v1.Normalize()等于将v1自身进行了单位化处理，v1变成了新的向量
        v1.Normalize();
        Debug.Log("v1.Normalize()后v1的值：" + v1);
    }
}
```

在这段代码的Start方法中，首先初始化了一个Vector3向量v1，然后分别使用实例属性normalized、静态方法Normalize和实例方法Normalize对v1进行单位化，并将结果打印出来，如图14-1所示，对结果的解释请参考代码注释。

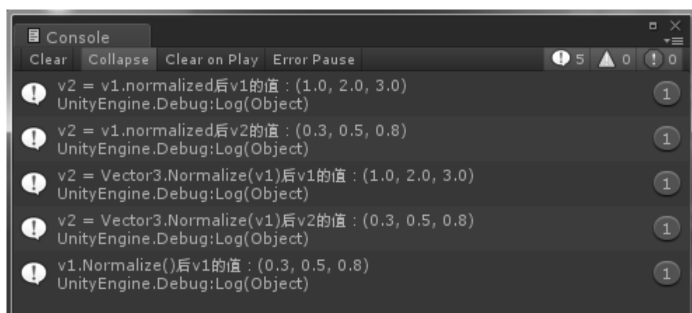


图14-1　实例演示运行结果

14.1.2　sqrMagnitude属性：模长平方

基本语法　public float sqrMagnitude { get; }

功能说明　此属性用于返回Vector3实例模长的平方值，即$x^2+y^2+z^2$的值。由于计算开方值比较消耗计算机资源，在非必需的情况下可以考虑用属性sqrMagnitude代替属性magnitude，例如比较两个向量长度的大小等。

实例演示　下面通过实例演示属性sqrMagnitude的使用。

```
using UnityEngine;
using System.Collections;

public class SqrMagnitude_ts : MonoBehaviour {
    void Start () {
        Vector3 v1 = new Vector3(3.0f,4.0f,5.0f);
```

```
            Vector3 v2 = new Vector3(1.0f, 2.0f, 8.0f);
            //属性magnitude用于求Vector3实例的模长
            Debug.Log("向量v1的长度："+v1.magnitude);
            Debug.Log("向量v2的长度："+v2.magnitude);
            float f1 = v1.sqrMagnitude;
            float f2 = v2.sqrMagnitude;
            Debug.Log("v1模长的平方值：" + f1 + "  v2模长的平方值：" + f2);
            if(f1 == f2){
                Debug.Log("向量v1和v2的模长一样大！");
            }
            else if (f1 > f2)
            {
                Debug.Log("向量v1的模长比较大！");
            }else{
                Debug.Log("向量v2的模长比较大！");
            }
        }
    }
```

在这段代码的Start方法中，首先初始化了两个Vector3变量v1和v2，然后分别使用属性magnitude和sqrMagnitude求向量v1、v2的模长和模长的平方值，最后打印出相关信息，程序运行输出结果如图14-2所示。

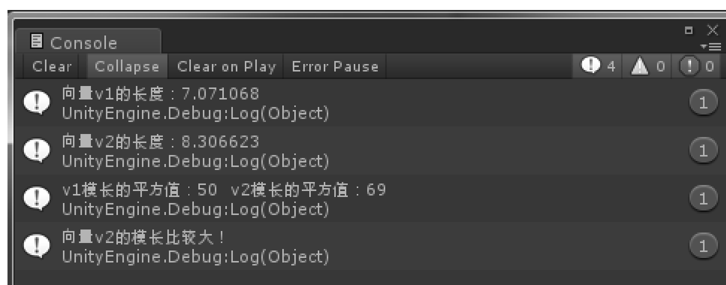


图14-2　实例属性sqrMagnitude的实例演示运行结果

14.2　Vector3 类实例方法

在Vector3类中涉及的实例方法只有Scale方法，由于Vector3的静态方法Scale与实例方法scale的功能相近，于是将它们放到一起介绍。

Scale方法：向量放缩

基本语法　public void Scale(Vector3 scale);

其中参数scale为参考向量。

功能说明　此方法可以对Vector3实例按参考向量scale进行放缩。注意此方法和静态方法Scale(a:Vector3, b:Vector3)的区别。设有三维向量v1=(x1,y1,z1)和v2=(x2,y2,z2)，则

- 执行代码v1.Scale(v2)后v2不变，v1的值变为：v1.x=x1*x2，v1.y=y1*y2，v1.z=z1*z2。
- 执行代码Vector3 v3= Vector3.Scale(v1, v2)后v1和v2不变，v3的值变为：v3.x=x1*x2，v3.y=y1*y2，v3.z=z1*z2。

实例演示　下面通过实例演示Vector3类的实例方法Scale和静态方法Scale的使用。

```
using UnityEngine;
using System.Collections;

public class Scale_ts : MonoBehaviour
{
    void Start()
    {
        Vector3 v1 = new Vector3(1.0f, 2.0f, 3.0f);
        Vector3 v2 = new Vector3(4.0f, 5.0f, 6.0f);
        //使用v1.Scale(v2)将使向量v1按向量v2进行放缩，无返回值
        v1.Scale(v2);
        Debug.Log("使用v1.Scale(v2)后v1的值：" + v1.ToString());
        Debug.Log("使用v1.Scale(v2)后v2的值：" + v2.ToString());
        //重设v1
        v1.Set(1.0f, 2.0f, 3.0f);
        //使用v3 = Vector3.Scale(v1, v2)将会返回向量v1按向量v2进行放缩后的向量v3
        //v1、v2不会改变
        Vector3 v3 = Vector3.Scale(v1, v2);
        Debug.Log("使用v3=Vector3.Scale(v1,v2)后v1的值：" + v1.ToString());
        Debug.Log("使用v3=Vector3.Scale(v1,v2)后v2的值：" + v2.ToString());
        Debug.Log("使用v3=Vector3.Scale(v1,v2)后v3的值：" + v3.ToString());
    }
}
```

在这段代码中，首先声明和实例化了两个Vector3变量v1和v2，然后分别演示了使用实例方法和类方法对Vector3实例进行放缩的使用方法。从输出结果（见图14-3）可以发现，在使用实例方法进行放缩后，变量v1的值改变了。但是在使用类方法放缩后，变量v1和v2的值没有改变，而是将放缩后的值被赋给了新的变量v3。实例方法无返回值，但会改变初始值，类方法不改变初始值，但会占用一个新的变量。这两种方法各有优劣，在程序开发中到底采用哪种方式需要根据实际情况而定。

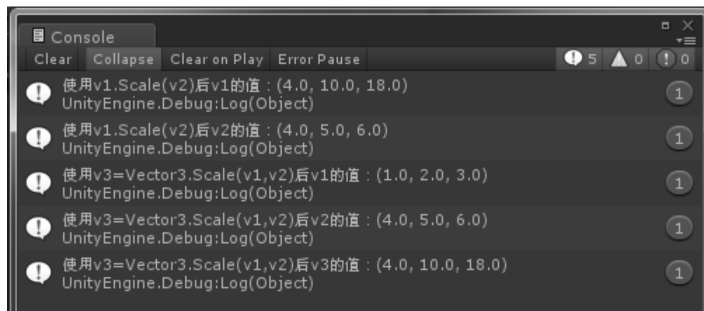


图14-3　实例演示运行结果

14.3 Vector3 类静态方法

在Vector3类中，涉及的静态方法有Angle方法、ClampMagnitude方法、Cross方法、Dot方法、Lerp方法、MoveTowards方法、OrthoNormalize方法、Project方法、RotateTowards方法、Scale方法、Slerp方法和SmoothDamp方法。由于OrthoNormalize的两个重载方法的功能及使用区别较大，因此分两次进行介绍。下面将详细介绍这些方法。

14.3.1 Angle方法：求两个向量夹角

基本语法 public static float Angle(Vector3 from, Vector3 to);

功能说明 此方法用于返回向量from和to的夹角，单位为角度，返回值的范围为[0,180]，且当from和to中至少有一个为Vector.zero时，方法返回值为90。

实例演示 下面通过实例演示Angle方法的使用。

```
using UnityEngine;
using System.Collections;

public class Angle_ts : MonoBehaviour
{
    void Start()
    {
        Debug.Log("1:" + Vector3.Angle(Vector3.right, -Vector3.right));
        Debug.Log("2:" + Vector3.Angle(Vector3.right, -(Vector3.right + Vector3.up)));
        Debug.Log("3:" + Vector3.Angle(Vector3.right, Vector3.zero));
    }
}
```

在这段代码的Start方法中，依次打印出了3种不同情况下Vector静态方法Angle的返回值，输出结果如图14-4所示。

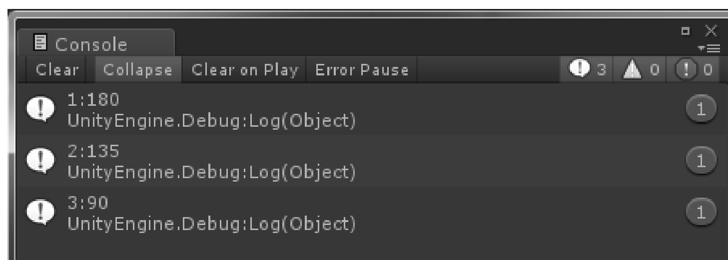


图14-4 方法Angle实例演示的运行结果

14.3.2 ClampMagnitude方法：向量长度

基本语法 public static Vector3 ClampMagnitude(Vector3 vector, float maxLength);

功能说明　此方法用于返回向量vector的一个同方向向量，其模长受maxLength的限制，对其使用说明如下。

- 返回向量的方向和vector方向相同。
- 当maxLength大于vector的模长时，返回向量与vector相同。
- 当maxLength小于vector的模长时，返回向量的模长等于maxLength，但方向与vector相同。

实例演示　下面通过实例演示方法ClampMagnitude的使用。

```
using UnityEngine;
using System.Collections;

public class ClampMagnitude_ts : MonoBehaviour {
    void Start () {
        Vector3 ts = new Vector3(1.0f,2.0f,3.0f);
        Debug.Log("ts的单位向量为："+ts.normalized+" ts模长为："+ts.magnitude);
        //当f值大于ts模长时，ClampMagnitude返回向量的模长为ts模长
        Debug.Log("当f值大于ts模长时：");
        float f = ts.magnitude + 1.0f;
        Vector3 ov = Vector3.ClampMagnitude(ts,f);
        Debug.Log("ov向量:" + ov.ToString()+"其单位向量为："+ov.normalized + " ov的模长为:" +
            ov.magnitude);
        //当f值小于ts模长时，ClampMagnitude返回向量的模长为f
        Debug.Log("当f值小于ts模长时：");
        f = ts.magnitude - 1.0f;
        ov = Vector3.ClampMagnitude(ts,f);
        Debug.Log("ov向量:" + ov.ToString() + "其单位向量为：" + ov.normalized + " ov的模长
            为:" + ov.magnitude);
    }
}
```

在这段代码的Start方法中，首先初始化了一个Vector3变量ts，然后分别演示了当变量f值大于ts模长和小于ts模长时方法ClampMagnitude的返回值，并打印出相关结果信息，如图14-5所示，对输出结果的解释请参考功能说明和代码注释。

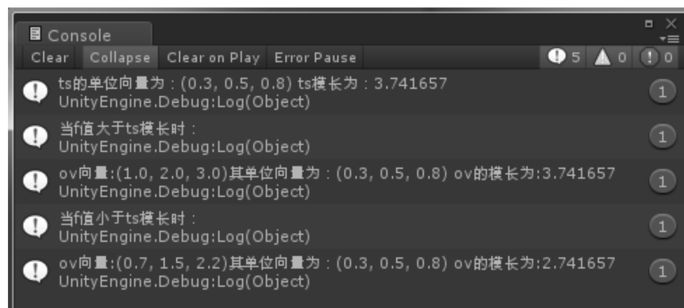


图14-5　静态方法ClampMagnitude实例演示的运行结果

14.3.3 Cross方法：向量叉乘

基本语法 public static Vector3 Cross(Vector3 lhs, Vector3 rhs);

功能说明 此方法用于求两个向量的叉乘。例如，设a、b和c均为Vector3实例，则执行程序代码

c=Vector3.Cross(a,b);

即为计算向量a和b的叉乘，即c=axb。设向量a与b的夹角为e，则有如下性质：

- ❑ c⊥a， c⊥b；
- ❑ 模长|c|=|a|*|b|sin(e)；
- ❑ a、b和c满足右手法则，即四指指向b的方向，然后向a的方向旋转，大拇指指向的方向就是c的方向，所以向量a、b和c是有一定次序关系的，即axb=–bxa。

实例演示 下面通过实例演示方法Cross的使用。

```
using UnityEngine;
using System.Collections;

public class Cross_ts : MonoBehaviour {
    public Transform A, B;
    Vector3 A_v, B_v;
    Vector3 C_v = Vector3.zero;

    void Update()
    {
        A_v = A.position;
        B_v = B.position;
        C_v = Vector3.Cross(A_v,B_v);
        //绘制从原点到各个坐标点的直线，以便观察
        Debug.DrawLine(Vector3.zero, A_v , Color.green);
        Debug.DrawLine(Vector3.zero, B_v, Color.yellow);
        Debug.DrawLine(Vector3.zero, C_v, Color.red);
    }
}
```

在这段代码中，首先声明了3个Vector3变量A_v、B_v和C_v，然后在Update方法中调用方法Cross将变量A_v和B_v的叉乘结果赋给变量C_v，最后分别绘制出从世界坐标系原点到向量A_v、B_v和C_v坐标点的直线。请自行运行程序在Scene视图而非Game视图中查看，拖动A和B的位置查看C_v的变化，图14-6是一张运行时截图及注释。

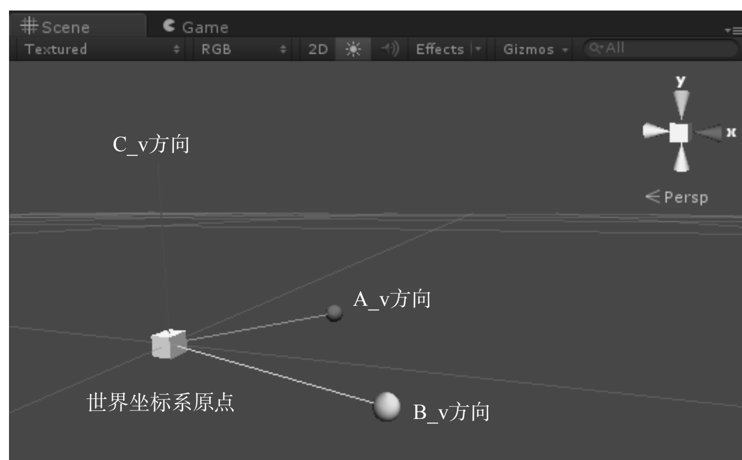


图14-6　静态方法Cross实例演示运行结果

14.3.4　Dot方法：向量点乘

基本语法　`public static float Dot(Vector3 lhs, Vector3 rhs);`

功能说明　此方法用于返回参数lhs和rhs的点乘。例如，设a和b均为Vector3实例，c为float类型数值，a与b的夹角度数为e，则执行程序代码

```
c=Vector3.Dot(a,b);
```

即为计算向量a和b的点乘，即c=a·b=|a|*|b|cos(e)。当返回值c>0时e∈(0,90)，当返回值c<0时e∈(90,180)。在实际开发中，通常利用点乘来确定两个物体的相对位置关系，例如敌人相对主角的位置关系。

实例演示　下面通过实例演示方法Dot的使用。

```
using UnityEngine;
using System.Collections;

public class Dot_ts : MonoBehaviour {
    public Transform A, B;
    Vector3 A_lv, AB_v;
    string str = "";

    void Update()
    {
        //将A的自身坐标系的forward向量转向世界坐标系中
        A_lv = A.TransformDirection(Vector3.forward);
        //A到B的差向量
        AB_v = B.position-A.position;
        float f = Vector3.Dot(A_lv, AB_v);
        if(f>0){
            str = "B在A自身坐标系的前方";
```

```
        }
        else if (f < 0)
        {
            str = "B在A自身坐标系的后方";
        }
        else {
            str = "B在A自身坐标系的正左方或右方";
        }
        //旋转A物体使得A的自身forward方向不断改变
        //虽然A、B的世界坐标都未改变，且在世界坐标系中A和B的位置关系没有改变
        //但在A的自身坐标系中B的相对位置在不断改变
        A.Rotate(new Vector3(0.0f,1.0f,0.0f));
}
//在界面上实时显示物体A、B的相对位置关系
    void OnGUI(){
        GUI.Label(new Rect(10.0f,10.0f,200.0f,60.0f),str);
    }
}
```

在这段代码中，首先声明了3个变量A_lv、AB_v和str，然后在Update方法中调用Dot方法，将向量A_lv和AB_v的点乘结果赋给局部变量f，最后通过f值来判断场景中物体A和B的相对位置关系。请自行运行程序查看，注意场景中物体A、B相对位置的变化。

14.3.5 Lerp方法：向量插值

基本语法 public static Vector3 Lerp(Vector3 from, Vector3 to, float t);

其中参数from为插值起始点坐标，参数to为插值结束点坐标，参数t为插值系数。

功能说明 此方法用于返回一个从参数from到to的线性插值向量。例如，设有Vector3实例A、B、C和float类型数值t，则当程序执行如下代码后

```
C=Vector3.Lerp(A,B,t);
```

- 当t≤0时，向量C=A；
- 当t≥1时，向量C=B；
- 当0<t<1时，向量C=A+(B−A)*t。

实例演示 下面通过实例演示方法Lerp的使用。

```
using UnityEngine;
using System.Collections;

public class Lerp_ts : MonoBehaviour
{
    public Transform start_T;//起始点位置物体
    public Transform end_T;//结束点位置物体
    Vector3 start_v, end_v;//起始和结束的两个Vector3
    float speed = 0.2f;//控制移动速度
```

14

```
    float last_time;//控制插值系数范围
    void Start()
    {
        start_v = start_T.position;
        end_v = end_T.position;
        last_time = Time.time;
    }
    void Update()
    {
        //利用插值改变物体位置坐标达到运动目的
        transform.position = Vector3.Lerp(start_v, end_v, (Time.time - last_time) * speed);
        if (transform.position == end_v)
        {
            //对调起始和结束点坐标
            transform.position = start_v;
            start_v = end_v;
            end_v = transform.position;
            transform.position = start_v;
            last_time = Time.time;
        }

    }
}
```

在这段代码中，首先声明了两个Vector3变量start_v和end_v，并在Start方法中对其初始化，然后在Update方法中调用方法Lerp，并将其返回值赋给transform的position，使得GameObject对象发生移动，请自行运行程序查看（物体做匀速往复运动）。

14.3.6　MoveTowards方法：向量插值

基本语法　public static Vector3 MoveTowards(Vector3 current, Vector3 target, float maxDistanceDelta);

功能说明　此方法用于返回一个从参数current到参数target的插值向量。例如，设有Vector3实例A=(ax,ay,az)、B=(bx,by,bz)和C=(cx,cy,cz)，向量A和B的差值为D(dx,dy,dz)，即D=B−A。sp为float类型值，则执行程序

C=Vector3. MoveTowards(A,B,sp);

后，向量C为：C=A+k*D，其中$k = sp > \sqrt{dx^2+dy^2+dz^2}\ ?\ 1 : \frac{sp}{\sqrt{dx^2+dy^2+dz^2}}$。

实例演示　下面通过实例演示方法MoveTowards的使用。

```
using UnityEngine;
using System.Collections;

public class MoveTowards_ts : MonoBehaviour {
    public Transform from_T, to_T;
```

```
    Vector3 from_v, to_v;
    Vector3 moves = Vector3.zero;
    float speed = 0.5f;

    void Start()
    {
        //初始化起始位置
        from_v = from_T.position;
        to_v = to_T.position;
    }

    void Update()
    {
        //在向量差值to_v-from_v的模长时间内slerps从from_v移动到to_v
        moves = Vector3.MoveTowards(from_v, to_v, Time.time * speed);
        //绘制从原点到slerps的红线，并保留100秒以便观察
        //运行时只能在scene视图中查看
        Debug.DrawLine(Vector3.zero, moves, Color.red, 100.0f);
    }
}
```

在这段代码中，首先声明了两个Vector3变量from_v和to_v，并在Start方法中对其初始化，然后在方法Update中将方法Vector3.MoveTowards的返回值赋给moves，最后绘制一条从世界坐标系原点到moves的直线。请自行运行程序查看moves点的移动轨迹，运行时请在Scene视图而非Game视图中查看，图14-7是一张运行时截图及注释。

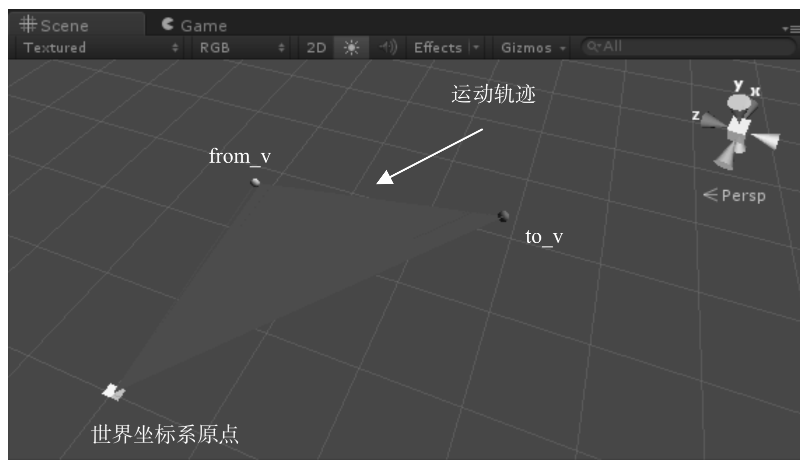


图14-7 静态方法MoveTowards实例演示运行结果

14.3.7 OrthoNormalize方法：两个坐标轴的正交化

基本语法 `public static void OrthoNormalize(ref Vector3 normal, ref Vector3 tangent);`

功能说明 此方法用于对向量normal进行单位化处理，并对向量tangent进行正交化处理。例如，

设有Vector3实例v1和v2（如图14-8所示），则执行如下程序代码后

```
Vector3. OrthoNormalize (ref v1, ref v2);
```

向量v1和v2都发生了改变。设向量v1在执行代码后值为v3，v2在执行代码后值为v4，则原始v1、v2和变换后的v3、v4的关系如图14-8所示，它们关系如下。

- 向量v3=v1.normalized；
- 向量v4与v3垂直，且v4模长为1；
- 向量v1、v2、v3和v4虽然都为三维向量，但它们在同一平面上。

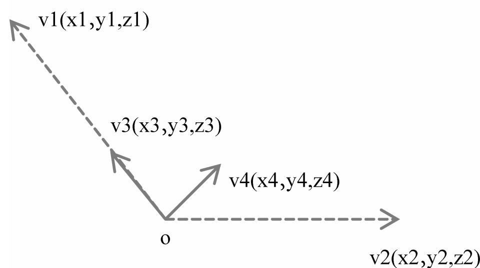


图14-8　OrthoNormalize示意图

实例演示　下面通过实例演示方法OrthoNormalize的使用。

```
using UnityEngine;
using System.Collections;

public class OrthoNormalize_ts : MonoBehaviour {
    public Transform one_T, two_T;
    Vector3 one_v, two_v;
    Vector3 one_l, two_l;

    void Start()
    {
        //初始化起始位置
        one_v = one_T.position;
        two_v = two_T.position;
        //记录初始化位置
        one_l = one_v;
        two_l = two_v;
    }

    void Update()
    {
        Vector3.OrthoNormalize(ref one_v,ref two_v);
        //绘制原始向量和OrthoNormalize处理后的向量
        Debug.DrawLine(Vector3.zero, one_l, Color.black);
        Debug.DrawLine(Vector3.zero, two_l, Color.white);
        Debug.DrawLine(Vector3.zero, one_v, Color.red);
        Debug.DrawLine(Vector3.zero, two_v, Color.yellow);
```

```
    }
}
```

在这段代码中，首先声明了4个Vector3变量，并在Start方法中对其初始化，然后在Update方法中调用方法OrthoNormalize，来对变量one_v和two_v进行正交化处理，最后根据正交化前后变量的值绘制出4条直线，请自行运行程序，在Scene视图而非Game视图中查看，图14-9是一张运行时截图及注释。

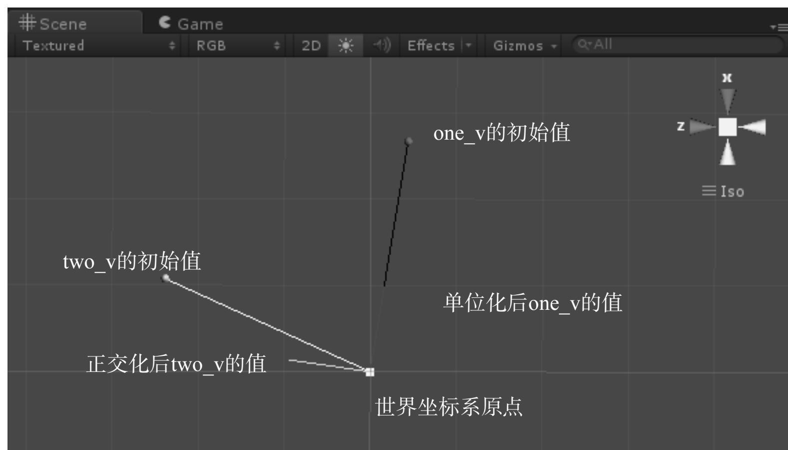


图14-9 静态方法OrthoNormalize的实例演示运行结果（两个参数）

14.3.8 OrthoNormalize方法：3 个坐标轴的正交化

基本语法 public static void OrthoNormalize(ref Vector3 normal, ref Vector3 tangent, ref Vector3 binormal);

功能说明 此方法用于对向量normal进行单位化处理，并对向量tangent和binormal进行正交化处理，对其使用说明如下。

- normal及tangent的功能和变化与方法OrthoNormalize (ref normal : Vector3, ref tangent : Vector3)中的相同，请参考其功能说明。
- 向量binormal垂直于由向量normal和tangent组成的平面，且向量binormal变换前后的夹角小于90度，即执行OrthoNormalize之后，binormal的方向可能垂直于由normal和tangent组成的平面的正面也可能是负面，到底垂直于哪个面由初始binormal的方向决定。

实例演示 下面通过实例演示方法OrthoNormalize的使用。

```
using UnityEngine;
using System.Collections;

public class OrthoNormalize2_ts : MonoBehaviour {
```

```
    public Transform one_T, two_T, three_T;
    Vector3 one_v, two_v, three_v;
    Vector3 one_l, two_l, three_l;

    void Start()
    {
        //初始化起始位置
        one_v = one_T.position;
        two_v = two_T.position;
        three_v = three_T.position;
        //保持初始值
        one_l = one_v;
        two_l = two_v;
        three_l = three_v;
    }

    void Update()
    {
        Vector3.OrthoNormalize(ref one_v, ref two_v,ref three_v);
        //绘制原始向量和OrthoNormalize处理后的向量
        Debug.DrawLine(Vector3.zero, one_l, Color.black);
        Debug.DrawLine(Vector3.zero, two_l, Color.white);
        Debug.DrawLine(Vector3.zero, three_l, Color.green);
        Debug.DrawLine(Vector3.zero, one_v, Color.red);
        Debug.DrawLine(Vector3.zero, two_v, Color.yellow);
        Debug.DrawLine(Vector3.zero, three_v, Color.blue);
    }
}
```

在这段代码中，首先声明了6个Vector3变量，并在Start方法中对其初始化，然后在Update方法中调用方法OrthoNormalize，对变量one_v、two_v和three_v进行正交化处理，最后根据正交化前后变量的值绘制出6条直线，请自行运行程序，在Scene视图而非Game视图中查看，图14-10是一张运行时截图及注释。

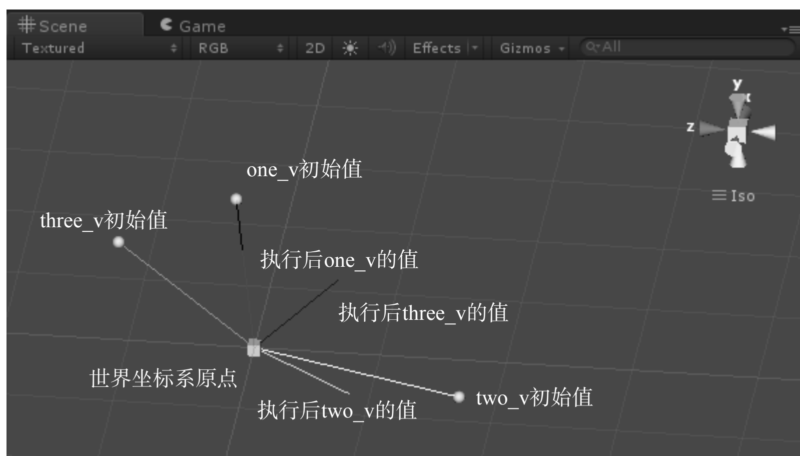


图14-10　静态方法OrthoNormalize的实例演示运行结果（3个参数）

14.3.9 Project方法：投影向量

基本语法 public static Vector3 Project(Vector3 vector, Vector3 onNormal);

功能说明 此方法用于返回向量vector在向量onNormal上的投影向量。如图14-11所示，执行以下程序代码

```
projects = Vector3.Project(from_T.position, to_T.position);
```

后，projects为from_T在to_T方向上的投影向量。另外，向量to_T无须为单位向量。

实例演示 下面通过实例演示方法Project的使用。

```
using UnityEngine;
using System.Collections;

public class Project_ts : MonoBehaviour {
    public Transform from_T, to_T;
    Vector3 projects = Vector3.zero;

    void Update()
    {
        projects = Vector3.Project(from_T.position, to_T.position);
        //绘制从世界坐标系原点到各个物体的直线
        Debug.DrawLine(Vector3.zero, projects, Color.black);
        Debug.DrawLine(Vector3.zero, from_T.position, Color.red);
        Debug.DrawLine(Vector3.zero, to_T.position, Color.yellow);
    }
}
```

在这段代码中，首先声明了两个Transform变量from_T和to_T，并初始化了一个Vector3变量projects，然后在Update方法中调用方法Project，求向量from_T.position在向量to_T.position上的投影向量，并将投影向量赋给变量projects。最后绘制从世界坐标系原点到各个物体坐标点及投影向量坐标点的直线。图14-11是一张运行时截图及注释，请自行运行程序，在Scene视图而非Game视图中查看。

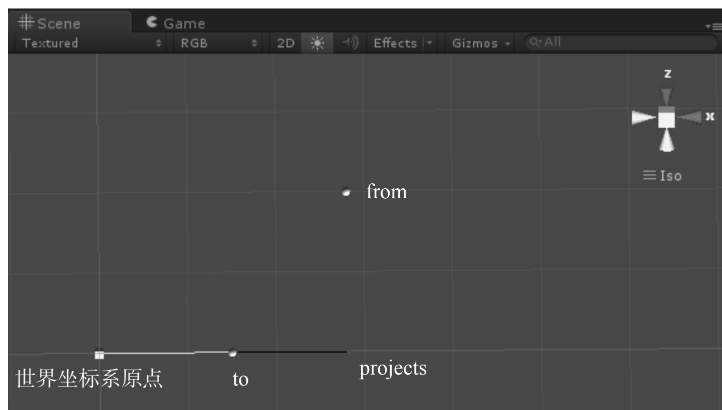


图14-11 静态方法Project实例演示运行结果

14.3.10　Reflect方法：反射向量

基本语法　public static Vector3 Reflect(Vector3 inDirection, Vector3 inNormal);

其中参数inDirection为入射向量，参数inNormal为镜面向量。

功能说明　此方法用于返回向量inDirection的反射向量，对其使用说明如下。

- 参数inNormal向量必须为单位向量，否则入射角和反射角不相等。
- 当inNormal取反时（即-inNormal），反射向量不受影响。
- 入射向量、反射向量和镜面向量共面。

实例演示　下面通过实例演示方法Reflect的使用。

```
using UnityEngine;
using System.Collections;

public class Reflect_ts : MonoBehaviour
{
    public Transform A, B;
    //A_v为镜面向量，B_v为入射向量
    Vector3 A_v, B_v;
    //R_v为反射向量
    Vector3 R_v = Vector3.zero;

    void Update()
    {
        //将镜面向量进行单位化处理，否则反射向量不一定为镜面反射
        A_v = A.position.normalized;
        B_v = B.position;
        R_v = Vector3.Reflect(B_v, A_v);

        Debug.DrawLine(-A_v * 1.5f, A_v *1.5f, Color.black);
        Debug.DrawLine(Vector3.zero, B_v, Color.yellow);
        Debug.DrawLine(Vector3.zero, R_v, Color.red);
    }
}
```

在这段代码中，首先声明了3个Vector3类型变量A_v、B_v和R_v，然后在Update方法中调用方法Reflect，将变量A_v和B_v的反射向量赋给变量R_v，最后分别绘制出镜面向量和从世界坐标系原点到向量A_v和B_v坐标点的直线。请自行运行程序，在Scene视图而非Game视图中查看，试着拖动A和B的位置查看R_v的的变化，图14-12是一张运行时截图及注释。

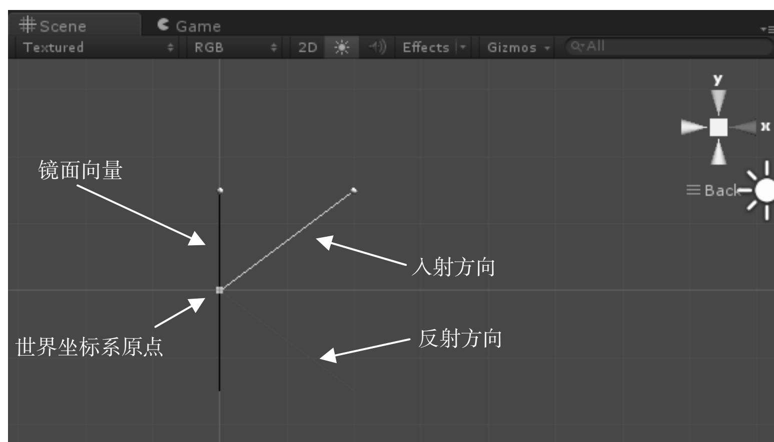


图14-12 静态方法Reflect实例演示运行结果

14.3.11 RotateTowards方法：球形插值

基本语法 public static Vector3 RotateTowards(Vector3 current, Vector3 target, float maxRadiansDelta, float maxMagnitudeDelta);

其中参数current为起始点坐标，参数target为目标点坐标，参数maxRadiansDelta为角度旋转系数，参数maxMagnitudeDelta为模长系数。

功能说明 此方法用于返回从参数current到target的球形旋转插值向量，此方法可控制插值向量的角度和模长。例如，设有Vector3实例A、B和C，有float类型数值R和L，向量A和B夹角的弧度为e，则执行以下程序代码后

```
C=Vector3. RotateTowards(A,B,R,L);
```

- 当$R\in[0,e]$时，向量C和A的弧度为R，即当R从0增加到e时，向量C与A的角度也会线性增加到e。向量C的模长为向量A的模长加上L的值，即$|C|=|A|+L$。
- 当$R<0$时，向量C会沿着从A到B的反方向旋转。
- 当$R>e$时，参数R以e来计算角度，即R大于e时C与B的方向相同。
- 总之，R值决定了向量C的方向，而L影响了C的模长，无论L取值多少，当$R>e$时向量C与B的方向总是相同的。
- 向量A、B和C在同一平面上。

实例演示 下面通过实例演示方法RotateTowards的使用。

```
public class RotateTowards_ts : MonoBehaviour
{
    public Transform from_T, to_T;
    Vector3 from_v, to_v;
    Vector3 rotates = Vector3.zero;
```

```
    float speed = 0.2f;
    float l;

    void Start()
    {
        //初始化起始位置
        from_v = from_T.position;
        to_v = to_T.position;
        //l取值为0时，rotates会以from_v的模长运动到to_v方向
        // l = 0.0f;
        //l取值为(to_v - from_v).sqrMagnitude时，rotates会以to_v的模长运动到to_v方向
        l = (to_v - from_v).sqrMagnitude;
    }

    void Update()
    {
        //在1/speed时间内rotates从from_v移动到to_v
        rotates = Vector3.RotateTowards(from_v, to_v, Time.time*speed,l);
        //绘制从原点到slerps的红线，并保留100秒以便观察
        //运行时只能在scene视图中查看
        Debug.DrawLine(Vector3.zero, rotates, Color.red, 100.0f);
    }
}
```

在这段代码中，首先声明了3个变量from_v、to_v和l，并在Start方法中对其初始化，然后在方法Update中将方法Vector3.RotateTowards的返回值赋给rotates，最后绘制一条从世界坐标系原点到rotates的直线。请自行运行程序，查看rotates点的移动轨迹，运行时请在Scene视图而非Game视图中查看，图14-13是一张运行时截图及注释。

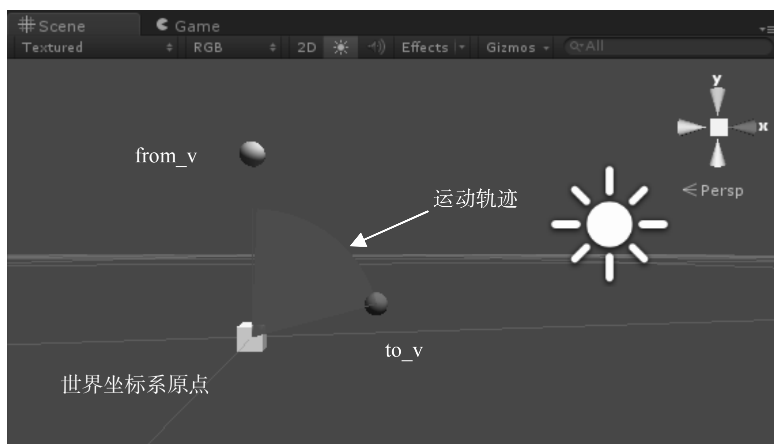


图14-13　静态方法RotateTowards实例演示运行结果

14.3.12　Scale方法：向量放缩

基本语法　`public static Vector3 Scale(Vector3 a, Vector3 b);`

功能说明 此方法用于返回向量a和b的乘积。注意此方法和实例方法Scale (scale : Vector3)的区别。例如，设有Vector3实例v1=(x1,y1,z1)和v2=(x2,y2,z2)，则：

- 执行代码v1.Scale(v2)后v2不变，v1的值变为v1.x=x1*x2，v1.y=y1*y2，v1.z=z1*z2。
- 执行代码Vector3 v3= Vector3.Scale(v1, v2)后v1和v2不变，v3的各个分量值变为v3.x=x1*x2，v3.y=y1*y2，v3.z=z1*z2。

实例演示 下面通过实例演示实例方法Scale和静态方法Scale的使用。

```
using UnityEngine;
using System.Collections;

public class Scale_ts : MonoBehaviour
{
    void Start()
    {
        Vector3 v1 = new Vector3(1.0f, 2.0f, 3.0f);
        Vector3 v2 = new Vector3(4.0f, 5.0f, 6.0f);
        //使用v1.Scale(v2)将使向量v1按向量v2进行放缩，无返回值
        v1.Scale(v2);
        Debug.Log("使用v1.Scale(v2)后v1的值：" + v1.ToString());
        Debug.Log("使用v1.Scale(v2)后v2的值：" + v2.ToString());
        //重设v1
        v1.Set(1.0f, 2.0f, 3.0f);
        //使用v3 = Vector3.Scale(v1, v2)将会返回向量v1按向量v2进行放缩后的向量v3
        //v1、v2不会改变
        Vector3 v3 = Vector3.Scale(v1, v2);
        Debug.Log("使用v3=Vector3.Scale(v1,v2)后v1的值：" + v1.ToString());
        Debug.Log("使用v3=Vector3.Scale(v1,v2)后v2的值：" + v2.ToString());
        Debug.Log("使用v3=Vector3.Scale(v1,v2)后v3的值：" + v3.ToString());
    }
}
```

在这段代码的Start方法中，首先初始化了两个变量v1和v2，然后分别演示了实例方法scale和静态方法scale对变量v1和v2的使用，并打印出相关信息，结果如图14-14所示，对运行结果的解释请参考代码注释。

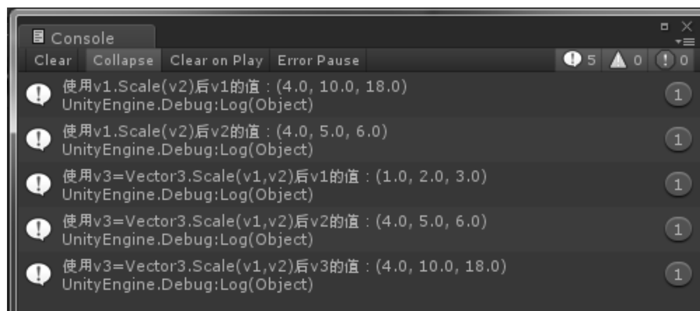


图14-14 静态方法Scale实例演示运行结果

14.3.13　Slerp方法：球形插值

基本语法　public static Vector3 Slerp(Vector3 from, Vector3 to, float t);

其中参数from为插值起始点坐标，参数to为插值结束点坐标，参数t为插值系数。

功能说明　此方法用于返回从参数from点到参数to点的球形插值向量。例如，设现有Vector3实例A和B（如图14-15所示），则执行如下程序代码后

```
Vector3 C=Vector3. Slerp (from, to, t);
```

- 当t≤0时，向量C=A；
- 当t≥1时，向量C=B；
- 当t从0增加到1时，向量C会从起始点A绕着A×B（即向量A和B的叉乘）的方向匀速移动到向量B，此处的匀速是指角度旋转的匀速，即向量C与B的夹角k=e*(1–t)，如图14-15所示，这样便可确定向量C的方向。而向量C的模长计算公式则为：

$$|C| = \sqrt{ax^2 + ay^2 + az^2} + \left(\sqrt{bx^2 + by^2 + bz^2} - \sqrt{ax^2 + ay^2 - az^2}\right) * t;$$

这样便可以确定向量C了。

- 当向量A和B中某个分量的值都为0时，比如它们的y轴分量都为0，即ay=by=0时，则A将绕着y轴在xz平面匀速旋转向B移动，并且在移动过程中C.y的值始终为0。

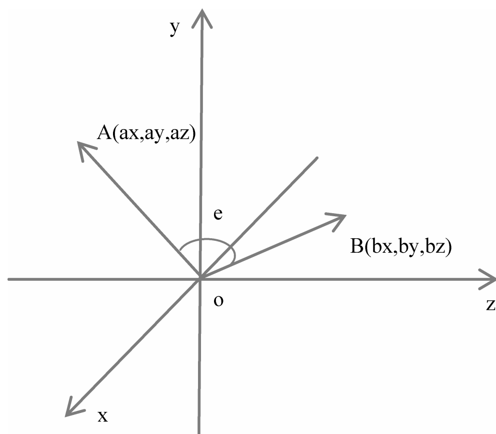


图14-15　Slerp球形插值

实例演示　下面通过实例演示方法Slerp的使用。

```
using UnityEngine;
using System.Collections;

public class Slerp_ts : MonoBehaviour {
    public Transform from_T, to_T;
```

```
    Vector3 from_v, to_v;
    Vector3 slerps = Vector3.zero;
    float speed = 0.1f;

  void Start () {
       //初始化起始位置
       from_v = from_T.position;
       to_v = to_T.position;
  }

  void Update () {
       //在1/speed时间内slerps从from_v移动到to_v
       slerps = Vector3.Slerp(from_v,to_v,Time.time*speed);
       //绘制从原点到slerps的红线，并保留100秒以便观察
       //运行时只能在scene视图中查看
       Debug.DrawLine(Vector3.zero,slerps,Color.red,100.0f);
  }
}
```

在这段代码中，首先声明了3个Vector3变量from_v、to_v和slerps，并在Start方法中对其初始化，然后在方法Update中将方法Vector3.Slerp的返回值赋给slerps，最后绘制一条从世界坐标系原点到slerps的直线。请自行运行程序，查看slerps点的移动轨迹，运行时请在Scene视图而非Game视图中查看，图14-16是一张运行时截图及注释。

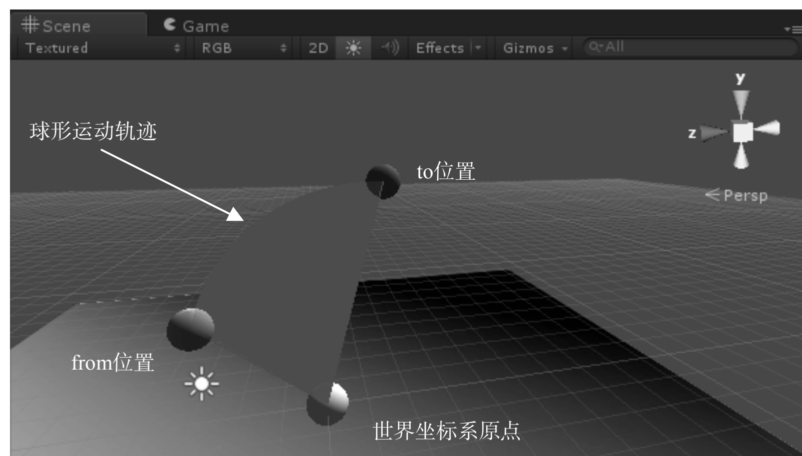


图14-16　静态方法Slerp的实例演示运行结果

14.3.14　SmoothDamp方法：阻尼移动

基本语法　(1) public static Vector3 SmoothDamp(Vector3 current, Vector3 target, ref Vector3 currentVelocity, float smoothTime);

(2) public static Vector3 SmoothDamp(Vector3 current, Vector3 target, ref Vector3 currentVelocity, float smoothTime, float maxSpeed);

(3) public static Vector3 SmoothDamp(Vector3 current, Vector3 target, ref Vector3 currentVelocity, float smoothTime, float maxSpeed, float deltaTime);

其中参数current为起点坐标，参数target为终点坐标，参数currentVelocity为当前帧移动向量，参数smoothTime为接近目标时的阻尼强度，参数maxSpeed为最大移动速度，默认值为无穷大，参数deltaTime为控制当前帧实际移动的距离，即为maxSpeed*deltaTime，默认值为Time.deltaTime。

功能说明 此方法用于模拟GameObject对象从current点到target点之间的阻尼运动。

实例演示 下面通过实例演示方法SmoothDamp的使用。

```
using UnityEngine;
using System.Collections;

public class SmoothDamp_ts : MonoBehaviour {
    public Transform from_T, to_T;
    public float smoothTime, maxSpeed,delta_time;
    Vector3 to_v;
    Vector3 speed = Vector3.zero;

    void Start()
    {
        //初始化起始位置
        transform.position = from_T.position;
        to_v = to_T.position;
        //初始化系数
        smoothTime = 1.5f;
        maxSpeed = 10.0f;
        delta_time = 1.0f;
    }

    void Update()
    {
        transform.position = Vector3.SmoothDamp(transform.position, to_v, ref speed,
smoothTime, maxSpeed, Time.deltaTime * delta_time);
    }
}
```

在这段代码中，首先声明了5个用于SmoothDamp方法的变量，并在Start方法中对其初始化，然后在Update方法中调用方法SmoothDamp，并将返回值赋给transform的position，用GameObject对象来模拟阻尼运动，请自行运行程序查看。

14.4 Vector3 类运算符

在Vector3类中，涉及的运算符主要有相等（“==”）运算符，下面简要介绍这个运算符。

operator == (lhs : Vector3, rhs : Vector3)

功能说明 此运算符用于判断向量lhs和rhs是否足够接近或相等。当参数中的向量lhs和rhs不相等

但足够接近时也会返回true，如实例演示所示。经本人电脑测试，当lhs和rhs的各个分量的小数点后五位相同时就可能返回true，当lhs和rhs的各个分量的小数点后六位都相同时则一定返回true。

实例演示　下面通过实例演示运算符“==”的使用。

```
using UnityEngine;
using System.Collections;

public class EqualOrNot_ts : MonoBehaviour
{
    void Start()
    {
        Vector3 v1 = new Vector3(1.123451f, 2.123451f, 3.123451f);
        Vector3 v2 = new Vector3(1.123452f, 2.123452f, 3.123452f);
        if (v1 == v2)
        {
            Debug.Log("v1==v2");
        }
        else
        {
            Debug.Log("v1!=v2");
        }
    }
}
```

在这段代码的Start方法中，首先初始化了两个Vector3变量v1和v2，向量v1和v2虽然不相同但分量值大小非常接近，然后使用运算符“==”来判断向量v1和v2是否相等，输出结果如图14-17所示。由输出结果可知，当两个向量足够接近时，运算符“==”的返回值会为true。

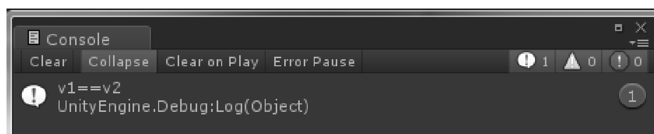


图14-17　静态方法operator ==实例演示运行结果

14.5　关于 Vector3.Lerp 和 Vector3.MoveTowards 的功能注解

- 相同点：它们的运动轨迹都为直线。
- 不同点：Lerp (from , to, t)方法中t的有效范围为[0,1]。当t<0时，参数t按0计算，当t>1时，参数t按1计算。当t≥1时，返回向量和向量to相同，t值的有效范围与from和to的取值无关。而方法MoveTowards (current, target, maxDistanceDelta)中maxDistanceDelta的有效范围为（ –∞,|target-current|]，当maxDistanceDelta与模长|target-current|相等时，返回向量才和target相同，即maxDistanceDelta的有效范围与current和target的取值有关。

14.6 关于 Vector3.RotateTowards 和 Vector3.Slerp 的功能注解

- 相同点：它们的移动轨迹都为弧形。
- 不同点：Slerp (from, to, t)方法的返回向量的模的大小是根据向量from和to的模的大小自动均匀放缩的，并且当t≥1时返回值坐标一定会和to相同。其返回向量的运动轨迹是个弧形但不一定为圆形，除非from和to的模长相等。

RotateTowards (current, target, maxRadiansDelta, maxMagnitudeDelta)方法的返回向量的模的大小是由current的模长和maxMagnitudeDelta共同决定的。如果在运动过程中current和maxMagnitudeDelta保持不变，则返回向量的模长将保持不变，即做匀速圆周运动。所以返回向量点的移动轨迹不能保证从current点到target点，但无论模长如何，当maxRadiansDelta大于或等于current和target夹角的弧度值时，返回向量的方向将和target相同。

第 15 章

游戏实例——坚守阵地

本章以一个游戏实例来描述使用Unity开发游戏的过程和一些API的用法。为了更好地说明实际游戏的开发过程以及对更多的API起到示范作用，本游戏中的一些脚本可能不是最优的。本章分为游戏概述、建模与导入、程序脚本和制作简单小地图4个部分。

15.1 游戏概述

本游戏是模拟第一人称的射击游戏，在游戏中玩家需要通过控制机枪来消灭不断进攻的坦克，当机枪的生命值低于0时游戏结束。

本游戏分为Game01、Game02和Game这3个场景，各个场景的截图如图15-1至图15-3所示。



图15-1　Game01场景截图

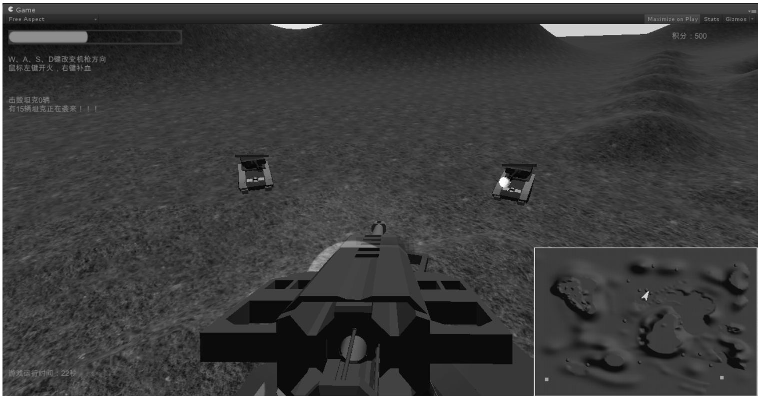


图15-2　Game02场景截图



图15-3　Game03场景截图

各个场景的功能说明如表15-1所示。

表15-1　游戏场景说明

场景名	说　　明
Game01	游戏开始界面，点击“Enter”键加载Game02场景
Game02	游戏中场景，在本场景中玩家通过键盘和鼠标来控制机枪消灭坦克，当机枪被坦克炮弹击毁（即生命值小于0）时本局游戏结束，进入Game03场景
Game03	游戏结束界面，点击“Enter”键加载Game02场景重新开始游戏，点击“Q”键退出游戏

游戏的操作说明如表15-2所示。

表15-2　游戏操作说明表

操　　作	说　　明
A	按下键盘上的“A”键，机枪会绕着y轴方向逆时针持续旋转，直到抬起“A”键结束
D	按下键盘上的“D”键，机枪会绕着y轴方向顺时针持续旋转，直到抬起“D”键结束
W	按下键盘上的“W”键，机枪会绕着x轴方向顺时针持续旋转，但其上仰角度不会超过30度
S	按下键盘上的“S”键，机枪会绕着x轴方向逆时针持续旋转，但其下仰角度不会超过25度
鼠标左键	每按下一次鼠标左键便会发射一颗子弹，如果子弹在5秒钟内未击中任何目标则自动销毁
鼠标右键	每按下一次鼠标右键补一次血，同时会消耗300积分。当积分低于300或机枪处于满血状态时点击无效

本游戏中用到的资源文件如图15-4所示，在Assets文件夹下有6个子文件夹，它们各自的内容如下。

- Detonator：一个爆炸效果插件。
- material：存放自建的材质球。
- prefabs：存放自建的预制组件。
- scripts：存放自建的脚本文件。
- something：存放一些原始资源，例如模型的FBX文件、贴图文件等。
- Standard Assets：系统自带的标准资源库。

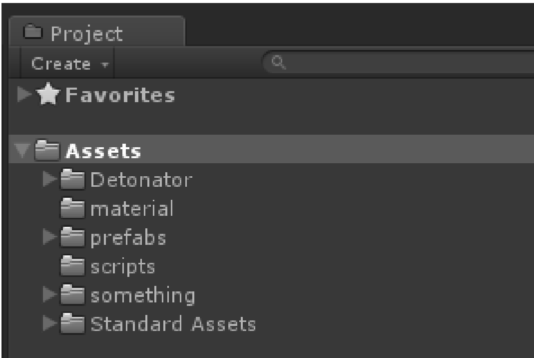


图15-4　游戏Project目录

15.2　建模与导入

一般而言，在游戏开发中，负责编程的人员和负责建模的人员一般不会是同一个人，这种情况下双方就需要相互沟通，以明确对方需要什么信息，使项目开发少走弯路。建模人员往往需要和编程人员确定模型应该怎样打组以及某些特殊部件的坐标轴应该怎么设置。一个合理的模型不仅能提高编程人员的开发效率，往往还会影响到游戏开发完后的运行效率。就本游戏而言，需要用到

的模型如表15-3所示。

表15-3　游戏模型说明

模型名称	说　　明
机枪模型	由于在操作机枪的时候枪管和底座是分开的，于是在对机枪模型打组的时候，可以分为枪管（pg）和底座（instance_1）两部分，如图15-5所示。当把机枪模型导入Unity后，需要再为其添加一个空对象（jq_kh_point）、一个跟随相机（M_Camera）和小地图标记（maptag），并把它们都放到枪管的下面作为其子类，如图15-5所示，其中jq_kh_point用来确定机枪子弹的实例化位置。这些做好后，把它们做成一个预制组件（prefabs）。另外需要注意的是，由于枪管是绕着底座旋转的，所以需要把枪管的坐标轴位置放到底座的正上方，如图15-6所示
坦克模型	坦克模型同样分为炮管（pg）和底座（zuojia）两部分，如图15-7所示。模型导入Unity后，同样需要为其添加一个空对象（pd_point）作为炮管的子类，此处放在了其子类的子类下面。另外，还需要为坦克添加一个血条模型和一个小地图标记，作为坦克的子类即可，如图15-7所示。其炮管坐标轴的位置如图15-8所示
兵工厂模型	兵工厂用来生产坦克，在实例化坦克时注意不能让坦克模型和兵工厂模型发生穿透。可以在模型中添加一个空对象来确定实例化坦克的坐标，也可在脚本中设置
机枪子弹模型	本游戏中以系统自带的球体作为机枪子弹模型，调整大小及材质后制成预制组件放于prefabs文件夹下
坦克炮弹模型	本游戏中以系统自带的球体作为坦克炮弹模型，调整大小及材质后制成预制组件放于prefabs文件夹下
坦克血条模型	血条的制作有多种方式，本游戏中用材质贴图的偏移来模拟坦克血量的变化。首先制作一张材质贴图，如图15-9所示，然后新建一个材质球（xt），并把血条贴图作为材质球的贴图，接着设置材质球Tiling的x值为0.5，如图15-10所示，最后把材质球拖到plane上即可。游戏中通过脚本控制贴图的偏移量来模拟坦克的血量变化

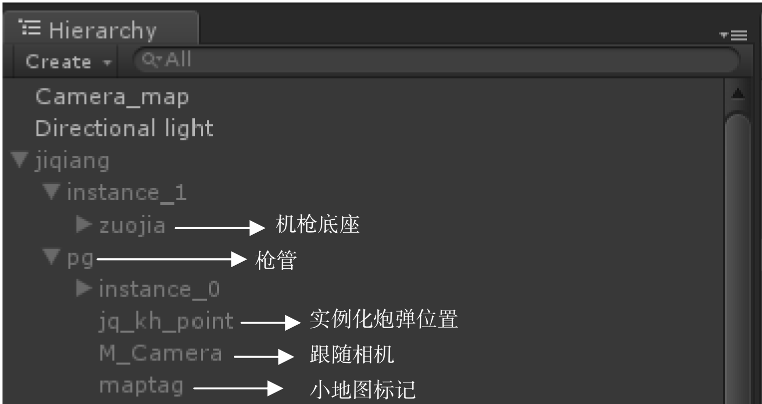


图15-5　机枪组成

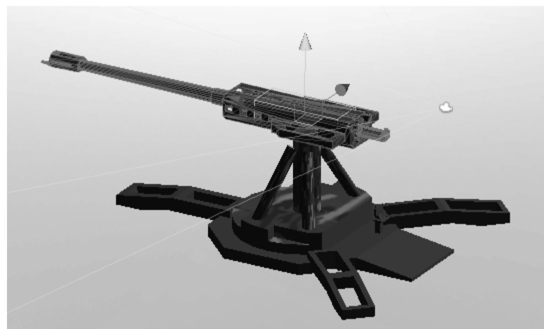


图15-6　机枪枪管坐标

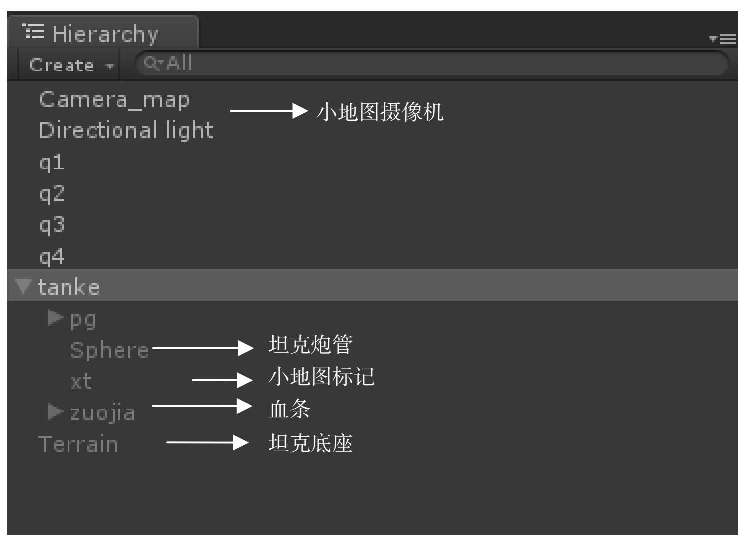


图15-7　坦克组成



图15-8　坦克炮管坐标



图15-9　血条贴图

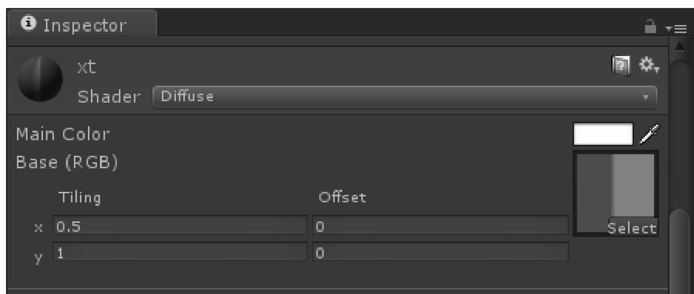


图15-10　血条材质球

模型导入完成后，还需要为它们添加一些组件，例如Collider、Rigidbody等，这里额外说明一下Collider的添加。对于从外部导入的模型，一般为其添加的Collider为Mesh Collider（Component→Physics→Mesh Collider），如图15-11所示，添加后要勾选Convex，否则可能会无效。对于一些比较复杂的模型，由于其自适应的碰撞器面数较多，因此会给程序运行带来较大的负担。对于一些精度要求不高的模型，可以为其添加比较简单的碰撞器，例如本游戏中为机枪炮管添加了一个简单的Box Collider，如图15-12所示。

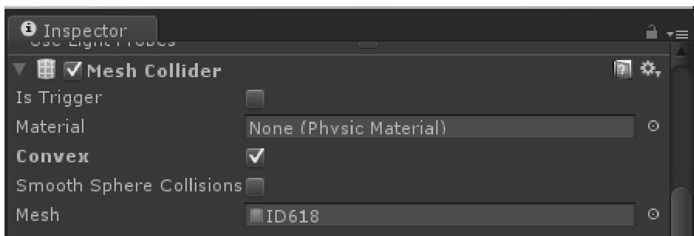


图15-11　坦克履带碰撞器

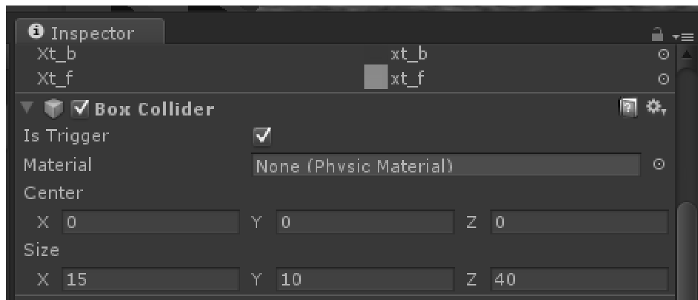


图15-12　机枪炮管碰撞器

15.3 程序脚本

本游戏中有9个脚本，它们的功能如表15-4所示。

表15-4 游戏脚本说明

脚本名称	脚本功能
Factory.cs	兵工厂脚本，用于坦克的实例化控制
Game01.cs	场景1的控制脚本，用于控制游戏的开始和新场景的加载
Game03.cs	场景3的控制脚本，用于控制游戏的重新开始和退出
Gamesetting.cs	游戏的设置脚本，用于控制游戏数据的加载、游戏对象的初始化以及一些全局控制变量的设置
Jiqiang.cs	机枪操作的控制脚本，用于控制机枪的旋转、开火及补血等
Jq_bullet.cs	机枪子弹脚本，用于控制机枪子弹的碰撞、爆炸及销毁
Set_game_data.cs	制作游戏数据的脚本，用于制作不同关卡中的游戏数据
Tk.cs	坦克控制脚本，用于控制坦克的所有行为
Tk_bullet.cs	坦克子弹脚本，用于控制坦克子弹的碰撞、爆炸及销毁

在编写脚本的过程中，为了示范更多的API，部分脚本代码写得比较烦琐，下面对部分脚本进行说明。

1. Factory.cs脚本

该脚本用于控制兵工厂对坦克的实例化。由于只有当当前游戏中坦克的数量小于坦克最大值时才需要实例化，所以脚本中去掉了Update方法，用InvokeRepeating方法来实现所需的功能。另外，为防止实例化的坦克模型和兵工厂模型发生穿透现象，需要对实例化的位置进行调整。该脚本的代码如下所示。

```
using UnityEngine;
using System.Collections;

/**
 * 兵工厂，用于生产坦克
 * */

public class Factory : MonoBehaviour
{
    public Transform tks;//声明坦克对象
    private Vector3 creat_tk_position;//声明坦克位置

    void Start()
    {
        //坦克实例化的位置与兵工厂的位置发生一定的偏移，以免模型之间发生穿透
        creat_tk_position = this.transform.position + new Vector3(10.0f, 4.0f, 10.0f);
        //游戏启动后2秒，开始调用方法creat_tk，以后每隔10秒调用一次此方法
        InvokeRepeating("creat_tk", 2.0f, 10.0f);
    }
```

```
        //实例化坦克
        private void creat_tk()
        {
            //当游戏中坦克数量少于游戏设置的最大坦克数量时，实例化一辆新坦克
            if (Gamesetting.num_tk < Gamesetting.tk_max_num)
            {
                Gamesetting.num_tk++;
                Instantiate(tks, creat_tk_position, Quaternion.identity);
            }
        }
    }
```

在这段代码中，首先声明了一个Transform类型的公共变量tks，用于指向游戏中坦克的预制组件，然后声明和实例化了坦克的实例化位置，最后根据Gamesetting中的tk_max_num值实例化相应数量的坦克。

2. Game01.cs脚本

该脚本用于控制游戏的开始和新场景的加载，对于新场景的加载，可以根据实际项目需求选择不同的方式加载，包括同步和异步加载方式。这里先说明一下，Game03.cs脚本与此脚本相似，具体内容请到源代码的工程项目中查看，在此不再赘述。Game01.cs脚本的代码如下所示。

```
    using UnityEngine;
    using System.Collections;
    /**
     * 场景1即游戏开始前的控制代码
     * */
    public class Game01 : MonoBehaviour
    {
        public Texture2D picture_bg;//背景图片
        public Texture2D progress_f, progress_b;//进度条前后图片
        float progress_length = 1;//进度条的实时长度
        bool is_press_enter = false;//是否按下enter键开始加载新场景
        float bg_x = 0.0f;//背景图片的x轴坐标，用于控制其向右移动
        float add_frame01 = 1.0f;
        float add_frame02 = 1.0f;
        bool is_load_over = false;

        //异步加载场景2，仅专业版可用
        //如果非异步加载，可以使用Application.LoadLevelAdditive(1);
        //异步加载通常用在加载资源较多比较消耗时间的情况下
        IEnumerator Start()
        {
            AsyncOperation async = Application.LoadLevelAdditiveAsync("Game02");
            //异步加载中
            Debug.Log("1:" + async.isDone);//是否加载完成
            Debug.Log("2:" + async.progress);//加载进度，范围0-1
            yield return async;
            //加载完成后
            Debug.Log("3:" + async.isDone);
            Debug.Log("4:" + async.progress);
            is_load_over = async.isDone;
```

```
    }

    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Return))
        {
            Debug.Log("您按下了Return键");
            //非异步加载索引值为1的场景，不改变当前场景的内容
            //Application.LoadLevelAdditive(1);
            is_press_enter = true;
        }
        if (is_press_enter)
        {
            progress_length += add_frame01;
            add_frame01++;
        }
        if (progress_length >= 500 && is_load_over)
        {
            is_press_enter = false;
            bg_x += add_frame02;
            add_frame02 += 3;
        }
        //当背景图片移出屏幕后，游戏开始
        if (bg_x > Screen.width)
        {
            //记录游戏开始时间
            Gamesetting.begin_time = Time.timeSinceLevelLoad;
            Gamesetting.which_step = 0;
            Destroy(this.gameObject);
        }
    }
    //绘制背景及进度条
    void OnGUI()
    {
        GUI.DrawTexture(new Rect(bg_x, 0.0f, Screen.width, Screen.height), picture_bg);
        if (is_press_enter)
        {
            GUI.DrawTexture(new Rect((Screen.width - 500) / 2, Screen.height - 50.0f, progress_length,
                15.0f), progress_f);
            GUI.DrawTexture(new Rect((Screen.width - 500) / 2, Screen.height - 50.0f, 500.0f, 15.0f),
                progress_b);
        }
    }
}
```

在这段代码中，演示了异步加载场景的方法，当玩家按下Enter键后开始加载新场景，通过控制progress_f的宽度值progress_length来模拟进度条的加载。

3. Gamesetting.cs脚本

该脚本用来控制游戏数据的加载、游戏对象的初始化以及一些全局控制变量的设置。实际游戏的开发通常要求场景中每个物体的参数都是可变的，以便使用相同的资源修改不同的参数来制作更

多的关卡。在本游戏的Game02原始场景中，只有地形、灯光和小地图摄像机，而其他对象（像机枪、坦克和兵工厂）都需要根据关卡数据进行临时的实例化。关卡数据的加载及对象的实例化代码片段如下所示。

```
/**
     * 读取相应关卡的初始化数据
     * path：读取文件的路径
     * name：读取文件的名称
     * _num:关卡值
     **/
    private void LoadFile(string path, string name, int _num)
    {
        string num = _num.ToString();
        string[] strs;
        //使用流的形式读取
        StreamReader sr = null;
        try
        {
            sr = File.OpenText(path + "//" + name);
            Debug.Log(path);
        }
        catch (System.Exception e)
        {
            //路径与名称未找到文件，则直接返回空
            //return null;
            Debug.Log("cann't find data file!" + e.Message);
        }
        string line = sr.ReadLine();
        do
        {
            strs = line.Split(',');
        } while (strs[0] != num && (line = sr.ReadLine()) != null);
        Debug.Log("guan qia zhi:" + strs[0]);
        //关闭流
        sr.Close();
        //销毁流
        sr.Dispose();

        //机枪的初始化位置
        string[] strs_child = strs[1].Split(' ');
        local_jq = new Vector3(float.Parse(strs_child[0]), float.Parse(strs_child[1]),
            float.Parse(strs_child[2]));
        //机枪的初始化生命值
        jq_values = int.Parse(strs[2]);
        //机枪实例化
        Instantiate(tran_jq, local_jq, Quaternion.identity);

        //初始实例化坦克数量
        num_tk = int.Parse(strs[3]);
        //坦克的初始化生命值
        tk_values = int.Parse(strs[4]);

        //兵工厂的初始化生命值
```

```
        factory_values = int.Parse(strs[6]);
        //兵工厂的初始化位置
        strs_child = strs[5].Split(' ');
        //兵工厂数量
        int tp = strs_child.Length / 3;
        factory_num = tp;

        Vector3[] local_factory = new Vector3[tp];
        for (int i = 0; i < tp; i++)
        {
            local_factory[i] = new Vector3(float.Parse(strs_child[i * 3]), float.Parse(strs_child
                [i * 3 + 1]), float.Parse(strs_child[i * 3 + 2]));
            //兵工厂实例化
            Instantiate(tran_factory, local_factory[i], Quaternion.identity);
        }
        //坦克数量最大值
        tk_max_num = int.Parse(strs[7]);
}
```

本实例游戏开始启动时，会根据关卡数据的num_tk值实例化相应数量的坦克，这些坦克的位置是随机分布在地形上的，但是要防止在实例化时出现一些不合常理的情况，例如坦克实例化的位置在山顶上，或实例化的位置离机枪位置太近。为防止坦克掉到山顶上，本程序先在地形上空向下（即负Y轴方向）发射一条射线，根据射线的长度来判断坦克着陆点是否合适，相关代码如下所示。

```
/**
     * 初始实例化坦克
     * */
    private void initial_tkVec()
    {
        Vector3 temp_vec;
        RaycastHit hit;
        Object go;

        Ray ray;
        float x, z;
        float _x, _z;
        //记录机枪的x、z坐标
        _x = tran_jq.position.x;
        _z = tran_jq.position.z;
        int count_temp = 0;

        while (count_temp < num_tk)
        {
            //获得在地图范围内的随机位置
            x = Random.Range(bj_pyl, ter_width - bj_pyl);
            z = Random.Range(bj_pyl, ter_length - bj_pyl);
            temp_vec = new Vector3(x, position_csh_y, z);
            //从随机确定的位置向-y轴发射一条射线
            ray = new Ray(temp_vec, new Vector3(0.0f, -1.0f, 0.0f));
            Physics.Raycast(ray, out hit, ter_height + 100);
            //判断位置是否合适
            //当位置距离机枪足够远，并且没落在山顶上时即认为位置合适
```

```
                if ((x - _x) * (x - _x) + (z - _z) * (z - _z) > length_to_jq && hit.distance > length_to_terrain)
                {
                    Vector3 v3 = hit.point;
                    v3.y = v3.y + 5;
                    //实例化坦克
                    go = Instantiate(tran_tk, v3, Quaternion.identity);
                    go.name = "tk_" + tk_cur_num;
                    tk_cur_num = tk_cur_num + 1;

                    count_temp++;
                }
            }
        }
```

本脚本中还有一些全局变量的设置，例如控制游戏启动变量which_step、机枪积分的实时值money、每局游戏坦克数量的最大值tk_max_num等，具体内容请查看项目工程代码。

4. Jiqiang.cs脚本

该脚本用于控制机枪的旋转、开火及补血等。机枪虽然可以进行左右360度的旋转，但其上下却不能做360度的旋转。在代码中，需要注意Input.GetKeyDown()和Input.GetKey()的区别，前者在按键按下时只响应一次，而后者在按键按下时会持续响应。在实例化子弹时，为了让子弹沿着枪管的方向飞行，需要设置其rotation为枪管自身的rotation，即transform.rotationd的值。Jiqiang.cs脚本的代码如下所示。

```
    void Update()
    {
        //W：机枪上转
        if (Input.GetKey(KeyCode.W) && angle_down_up < 30.0f)
        {
            Debug.Log("您按下了W键" + angle_down_up);
            angle_down_up += frame_angel_du;
            //机枪上下旋转的参考坐标系为自身坐标系，即默认值space.self
            transform.Rotate(Vector3.right, -frame_angel_du);
        }
        //S：机枪下转
        if (Input.GetKey(KeyCode.S) && angle_down_up > -25.0f)
        {
            Debug.Log("您按下了S键" + angle_down_up);
            angle_down_up -= frame_angel_du;
            transform.Rotate(Vector3.right, frame_angel_du);
        }
        //A：机枪左转
        if (Input.GetKey(KeyCode.A))
        {
            Debug.Log("您按下了A键");
            //机枪左右旋转的参考坐标系要为世界坐标系
            //否则在上下旋转后再进行左右旋转时会变形
            transform.Rotate(Vector3.up, -frame_angel_lr, Space.World);
        }
        //D：机枪右转
        if (Input.GetKey(KeyCode.D))
```

```
        {
            Debug.Log("您按下了D键");
            transform.Rotate(Vector3.up, frame_angel_lr, Space.World);
        }
        //鼠标左键开火
        if (Input.GetMouseButtonDown(0))
        {
            Debug.Log("您按下了鼠标左键");
            Instantiate(jq_pd, jq_kh_point.position, transform.rotation);
        }
        //鼠标右键补血
        if (Input.GetMouseButtonDown(1))
        {
            Debug.Log("您按下了鼠标右键");
            if (Gamesetting.money >= 300 && current_value < 300)
            {
                Gamesetting.money -= 300;
                current_value = Gamesetting.jq_values;
                xt_length = ((float)current_value / Gamesetting.jq_values) * 300.0f;
            }
        }
    }
```

在这段代码中，通过调用Input.GetKey方法控制机枪的上下左右旋转，并通过Input.GetMouseButtonDown方法控制机枪的开火及补血。

5. Jq_bullet.cs脚本

该脚本用于控制机枪子弹的碰撞、爆炸及销毁，其代码如下所示。Tk_bullet.cs脚本用于控制坦克炮弹的碰撞、爆炸及销毁，其代码与此相似，具体内容请到源代码的工程项目中查看，在此不再赘述。

```
using UnityEngine;
using System.Collections;

/**
 * 机枪子弹
 * */

public class Jq_bullet : MonoBehaviour
{
    public Transform tk_exp;//爆炸效果组件
    float this_time = 0.0f;
    void Start()
    {
        this_time = Time.time;
    }
    void Update()
    {
        //如果子弹4秒内未打到物体，则自动销毁，避免内存浪费
        if (Time.time - this_time > 4.0f)
        {
            Destroy(this.gameObject);
        }
```

```
    }

    void FixedUpdate()
    {
        //子弹以每秒100米的速度向局部forward方向运动
        transform.rigidbody.velocity = transform.TransformDirection(Vector3.forward * 100);
    }

    void OnTriggerEnter(Collider otherObject)
    {
        //爆炸效果及销毁对象
        Instantiate(tk_exp, transform.position, Quaternion.identity);
        Destroy(this.gameObject);
    }
}
```

在这段代码中，使用Time.time控制子弹的最长生存时间，用`transform`的`TransformDirection`方法控制子弹的飞行方向，并在`OnTriggerEnter`方法中实例化爆炸组件和控制子弹的销毁。

6. Set_game_data.cs脚本

该脚本用于制作不同关卡中的游戏数据，这些数据包括机枪实例化的位置、机枪的生命值、初始化坦克的数量等。运行此脚本便可以在项目工程的Assets目录下生成关卡数据文件。需要注意的是，当把工程导出为可执行文件时，需要把关卡数据文件复制到游戏的资源文件夹中，否则游戏运行时会因找不到数据文件而无法实例化游戏场景。Set_game_data.cs脚本的代码如下所示。

```
using UnityEngine;
using System.Collections;
using System.IO;
using System.Text;
//设置和生成游戏关卡数据
public class Set_game_data : MonoBehaviour
{
    StringBuilder sb = new StringBuilder("", 256);
    //关卡值
    string num = "1";
    //机枪位置
    string jiqiang = "";
    //机枪生命值
    string jqsmz = "222";
    //初始化坦克数量
    string tks = "12";
    //坦克生命值
    string tkz = "111";
    //兵工厂位置
    string bgc = "";
    //兵工厂生命值
    string bgcz = "333";
    //最大坦克数量
    string max_tk_num = "15";

    void Start()
```

```
{
    getDate();
    appends();
    //路径、文件名、信息
    CreateFile(Application.dataPath, "FileName", sb.ToString());
}
//一般而言，机枪和兵工厂的位置需要在地图上选取合适的位置
//制作关卡游戏数据之前，请先把机枪和兵工厂拖拽到地图上的合适位置
private void getDate()
{
    GameObject fsj1 = GameObject.FindGameObjectWithTag("jq");
    GameObject[] bgc1 = GameObject.FindGameObjectsWithTag("bgc");

    GameObject go;

    for (int i = 0; i < bgc1.Length; i++)
    {
        go = bgc1[i];
        bgc += go.transform.position.x + " " + go.transform.position.y + " " +
             go.transform.position.z;
        if (i != bgc1.Length - 1)
        {
            bgc += " ";
        }
    }

    jiqiang = fsj1.transform.position.x + " " + fsj1.transform.position.y + " " + fsj1.transform.
        position.z;
}
//拼接数据
private void appends()
{
    sb.Append(num);
    sb.Append(",");
    sb.Append(jiqiang);
    sb.Append(",");
    sb.Append(jqsmz);
    sb.Append(",");
    sb.Append(tks);
    sb.Append(",");
    sb.Append(tkz);
    sb.Append(",");
    sb.Append(bgc);
    sb.Append(",");
    sb.Append(bgcz);
    sb.Append(",");
    sb.Append(max_tk_num);
}

/**
* path：文件创建目录
* name：文件的名称
* _info：写入的内容
*/
```

```
        void CreateFile(string path, string name, string info)
        {
            //文件流信息
            StreamWriter sw;
            FileInfo t = new FileInfo(path + "//" + name);
            //输出数据存放路径
            Debug.Log(path);
            if (!t.Exists)
            {
                //如果此文件不存在，则创建一个新的文件
                sw = t.CreateText();
            }
            else
            {
                //如果此文件存在，则打开
                sw = t.AppendText();
            }
            //以行的形式写入信息
            sw.WriteLine(info);
            //关闭流
            sw.Close();
            //销毁流
            sw.Dispose();
        }
    }
```

在这段代码中，首先声明了很多需要自定义的变量，并在方法getDate()中获取一些必要的数据，请参考代码注释，然后在appends()方法中拼接数据，最后在CreateFile()方法中将数据输出到指定的文件中。

7. Tk.cs脚本

该脚本用于控制坦克的所有行为。坦克的所有行为可以用一个简单的有限状态机来表示，如表15-5所示，第一列为转换前状态名称，第一行为转换后状态名称。

表15-5　坦克行为的有限状态转换表

转换状态	游戏未开始	调整方向	前　进	调整炮管	发射炮弹
游戏未开始		游戏开始			
调整方向			方向调整完毕		
前进		前进遇阻		机枪进入坦克射程内	
调整炮管					炮管调整完毕
发射炮弹				坦克发生位移	

坦克调整方向的代码片段如下所示。

```
    //调整方向
        private void step_zero()
        {
```

```
            Vector3 vt = Vector3.MoveTowards(transform.forward, (tk_aim - transform.position).normalized,
                Time.time * 0.001f);
            float _x = (v3_temp.x - vt.x) * (v3_temp.x - vt.x) + (v3_temp.z - vt.z) * (v3_temp.z - vt.z);

            v3_temp = vt;
            vt = vt.normalized;
            zero_count++;
            if (_x != 0.0f && zero_count < zero_count_max)
            {
                transform.forward = vt;
            }
            else
            {
                which_step = 1;
                turnRightOrLeft();
                zero_count = 0;
            }
    }
```

在这段代码中，用Vector3的MoveTowards方法将坦克transform的forward方向转到指向机枪所在位置的方向。当单帧旋转角度为0时停止旋转，坦克开始前进。

当坦克角度偏转过大或单位时间内位移过小时都认为坦克遇阻，需要调整坦克的前进方向。前进代码片段如下所示。

```
    //前进-update
        private void step_first_update()
        {
            //坦克侧翻时需要矫正
            if (this.transform.up.y < 0)
            {
                this.transform.up = new Vector3(this.transform.up.x, 2.0f-this.transform.up.y,
                    this.transform.up.z);
            }

            float f1=Vector3.Angle(transform.forward,Vector3.up);
            float f2=Vector3.Angle(transform.right,Vector3.up);
            //当坡度太陡时认为坦克遇阻，此处z轴偏移不得大于44度，x轴偏移不得大于30度
            if(f1<46.0f||f1>134||f2<60||f2>120){
                first_is_stop = true;
            }
            //坦克每隔first_correct_count_max帧调整一次前进方向
            if (first_correct_count < first_correct_count_max)
            {
                first_correct_count++;

            }
            else
            {
                first_go_v3 = tk_aim - this.transform.position;
                first_go_v3 = (first_go_v3.normalized - this.transform.forward) / 64;
                first_is_rotate_count = 64;
                first_correct_count = 0;
```

```
            first_is_stop = false;
            first_is_stop_count = 0;
            first_last_count = 0;
            first_last_point = first_current_point;
        }
        //调整方向
        if (first_is_rotate_count > 0)
        {
            this.transform.forward += first_go_v3;
            first_is_rotate_count--;
            first_correct_count = 0;
        }
        //遇阻后方向调整
        if (first_is_stop)
        {
            if (first_is_turn_right)
            {
                transform.Rotate(Vector3.up, first_frame_angle, Space.Self);
            }
            else
            {
                transform.Rotate(Vector3.down, first_frame_angle, Space.Self);
            }
            first_is_stop_count++;
            //调整角度大于50度后结束
            if (first_is_stop_count * first_frame_angle > 50)
            {
                first_is_stop = false;
                first_is_stop_count = 0;
                first_last_count = 0;
                first_last_point = first_current_point;
            }
            first_correct_count = 0;
        }
    }

    //前进-FixedUpdate
    private void step_first_fixedUpdate()
    {
        //如果坦克未遇阻
        if (!first_is_stop)
        {
            //每隔first_check_wait时间检测一次坦克前进距离
            if (first_last_count < first_check_wait)
            {
                first_last_count++;
                transform.Translate(transform.forward * tk_speed, Space.World);
            }
            else
            {
                first_current_point = transform.position;
                first_len_last = first_current_point - first_last_point;
                //当前进距离过小时认为坦克遇阻
```

```
                if (first_len_last.x * first_len_last.x + first_len_last.z * first_len_last.z < min_len)
                {
                    first_is_stop = true;
                }
                else
                {
                    first_is_stop = false;
                    first_last_count = 0;
                    first_last_point = first_current_point;
                    first_is_stop_count = 0;
                }

            }
        }
    }
```

这段代码用来控制坦克的自动前进，分别处理了以下几种情况：

- 当坦克侧翻时即transform的up方向为负时需要矫正；
- 当坦克遇上过于陡峭的坡度时需要调整方向；
- 当坦克在一定时间内前进的距离过小时（例如遇到墙壁之类的障碍物），认为坦克遇阻，需要调整方向；
- 当坦克在未遇阻情况下每前进一段时间，需要检验一次前进方向。

当机枪进入坦克的有效射程时，坦克停止前进，需要调整炮管瞄准机枪，调整炮管的代码片段如下所示。

```
//调整炮管
    private void step_second()
    {
        Vector3 vt = Vector3.MoveTowards(tk_pg.transform.forward, (tk_pgaim -
            tk_pg.transform.position).normalized, Time.time * 0.001f);

        float _x = (v3_temp.x - vt.x) * (v3_temp.x - vt.x) + (v3_temp.z - vt.z) * (v3_temp.z - vt.z);
//当相邻两帧不再有角度调整时，炮管调整结束
        if (_x != 0.0f)
        {
            v3_temp = vt;
            tk_pg.transform.forward = vt.normalized;
        }
        else
        {
            v3_temp = Vector3.zero;
            which_step = 3;
        }
    }
```

在这段代码中，使用`Vector3.MoveTowards`方法来调整坦克炮管的旋转，直到坦克炮管瞄准机枪，即当相邻两帧不再有角度调整时，炮管调整结束，接下来坦克就向机枪发射炮弹，试图摧毁机枪。

发射炮弹时，需要注意实例化子弹的`rotation`值，为了让炮弹沿着炮管方向飞行，需要用炮管的

rotation值即tk_pg.transform.rotation作为炮弹的rotation值。发射炮弹的代码片段如下所示。

```
//发射炮弹
    private void step_third()
    {
        if (third_time < tk_fire_wait)
        {
            third_time += Time.deltaTime;
        }
        else
        {
            //实例化炮弹
            Instantiate(tk_bullet, tk_fire_point.position, tk_pg.transform.rotation);
            third_time = 0.0f;
        }
}
```

这段代码用于控制坦克炮弹的发射，坦克每隔tk_fire_wait秒发射一次炮弹，用tk_fire_point.position作为实例化炮弹的位置，用tk_pg.transform.rotation作为实例化炮弹的rotation值。

当坦克受到机枪攻击后会掉血，有关坦克血条的代码片段如下所示。

```
//计算血条值
    private void caculate_xt_value()
    {
        xt_value = ((float)tk_value / Gamesetting.tk_values) * 0.5f;
        xt_value = xt_value > 0.5f ? 0.5f : xt_value;
        tk_xt.renderer.material.SetTextureOffset("_MainTex", new Vector2(xt_value, 0.0f));
    }
    //计算血条的朝向，使血条始终朝向摄像机
    private void caculate_xt_foward()
    {
        Vector3 dot = Vector3.zero;
        dot.x = Camera.main.transform.eulerAngles.x - 90.0f;
        dot.y = Camera.main.transform.eulerAngles.y;
        tk_xt.transform.eulerAngles = dot;
}
```

这段代码用来控制血条的值和血条的朝向，此处使用材质的纹理偏移（material.SetTextureOffset）来模拟坦克血条的变化。为使血条始终朝向摄像机，需要使得血条的y轴欧拉角与摄像机的y轴eulerAngles相等，再调节血条的x轴欧拉角，使血条始终朝向摄像机。

15.4　制作简单小地图

接下来为本实例制作一个简单的小地图，以便于玩家在游戏运行过程中实时观察周围的情况。首先添加一个用于显示小地图的摄像机，命名为Camera_map，然后对Camera_map设置（如图15-13所示）如下。

(1) 设置ClearFlags为Depth Only，以便于消去小地图的杂边；

(2) 设置CullingMask为maplayer和terrainlayer，以便于按层有选择的渲染；

(3) 设置投影矩阵（Projection）为正交投影（orthographic）；

(4) 设置Viewport Rect为适当值，使得小地图位于右下角；

(5) 设置Depth值为0，其值要大于跟随相机的Depth值。



图15-13　小地图摄像机Camera_map的设置

然后把坦克、机枪和兵工厂需要在小地图中显示的部分（即模型中的maptag物体）的层选择为maplayer，而在跟随相机（jiqiang→pg→M_Camera）中，CullingMask选项不要选择maplayer。另外，需要删除在小地图中显示部件的Collider组件（如果存在的话），以免发生错误碰撞，例如坦克模型、机枪模型中的小地图标记中的Collider组件。

最后需要说明的是，当把项目导出为单独的exe文件时，需要把关卡数据文件“FileName”复制到exe的资源文件夹下，否则程序在加载场景Game02时无法运行。

Unity API 解析

Unity引擎的应用领域日益广泛，然而Unity具有上手快、提高难的特性，其中API就是阻碍新手能力提高的一大障碍。对API的熟悉程度直接影响着程序的开发效率，熟悉API也成了新手进阶的必经之路。

本书挑选了Unity引擎里一些核心API类（例如Object、GameObject、Rigidbody、Transform、Camera、Quaternion、Vector3等），进行了解析说明，解析内容包括API的使用方法、算法分析、边界条件、参数间的制约关系及注意事项等各个方面，对功能相近或使用方法相似的API进行了较为详细的比较说明。本书每个API都配有实例演示，便于读者理解和使用，是一本让Unity新手摆脱对API一知半解、快速提升能力的首选之书。

图灵社区：iTuring.cn
热线：(010) 51095186 转 600

分类建议 计算机/移动开发/游戏开发

人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-36651-1

定价：49.00元

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们:

微博 @图灵教育 : 好书、活动每日播报
微博 @图灵社区 : 电子书和好文章的消息
微博 @图灵新知 : 图灵教育的科普小组
微信 图灵访谈 : ituring_interview，讲述码农精彩人生
微信 图灵教育 : turingbooks